ROZDZIAŁ DWUDZIESTY PIĄTY: **OPTYMALIACJA NASZYCH PROGRAMÓW**

25.0 WSTĘP

Ponieważ optymalizacja programu jest generalnie jednym z ostatnich kroków w projektowaniu oprogramowania, jest to tylko przymiarka do omówienia optymalizacji programu w ostatni rozdziale tej książki. Przeszukując inne teksty, które omawiają ten temat, znajdziemy szeroką gamę opinii na ten temat. Niektóre teksty i artykuły ignorują zbiór instrukcji całkowicie i koncentrują się na znajdowaniu lepszych algorytmów. Inne dokumenty zakładają, że już znaleziono najlepszy algorytm i omawiają sposoby wybierania „najlepszej" sekwencji instrukcji realizujących tą pracę. Inne rozważają architekturę CPU i opisują jak „liczyć cykle" i pary instrukcji (zwłaszcza na procesorach superskalarnych lub procesorach z potokami) tworząc szybciej wykonujący się kod. Inne jeszcze rozpatrują architekturę systemu, nie tylko CPU, kiedy próbują zdecydować jak optymalizować mamy nasz program. Niektórzy autorzy spędzają dużo czasu wyjaśniając, że ich metoda jest „jedynie słuszną" dla przyspieszenia programu. Inni jeszcze odbiegają od inżynierii oprogramowania i zaczynają mówić o tym jak czas spędzony nad optymalizacja programu nie jest wart tego z różnych powodów. Cóż, rozdział ten nie ma zamiaru przedstawiać „jedynie słusznego sposobu", ani też spędzać wiele czasu na sprzeczaniu się o pewne techniki optymalizacji. Po prostu przedstawi kilka przykładów, opcji i sugestii. Ponieważ jesteś sobą po tym rozdziale, czas na Ciebie abyś zaczął podejmować samodzielne decyzje. Miejmy nadzieję, że rozdział ten dostarczy stosownych informacji dal podejmowania poprawnych decyzji.

25.1 KIEDY OPTYMALIZOWAĆ,KIEDY NIE OPTYMALIZOWAĆ

Proces optymalizacji nie jest tandetny Jeśli projektujemy program a potem stwierdzimy, że jest zbyt wolny, może będziemy musieli przeprojektować i napisać na nowo główną część tego programu dal uzyskania akceptowalnej wydajności. W oparciu o powyższe świat dzieli się na dwa obozy – tych , którzy optymizują wcześnie i tych , którzy optymalizują później. Obie grupy mają dobre argumenty; obie grupy mają jakieś złe argumenty. Przyjrzymy się obu stronom tych argumentów.

Grupa „optymalizujących później" (OL) używa argumentu 90/10: 90% czasu wykonywania programu jest spędzanych w 10% kodu. Jeśli próbujemy optymalizować każdy kawałek kodu jaki napisaliśmy (to jest, optymalizujemy kod zanim dowiemy się ,że musi być zoptymalizowany), 90% wysiłku włożonego pójdzie na marne. Z drugiej strony, jeśli najpierw piszemy kod w zwykły sposób a potem idziemy w optymalizację, możemy poprawić wydajność naszego programu przy mniejszym wkładzie pracy. W końcu jeśli zupełnie usuniemy 90% naszego programu, nasz kod będzie działał tylko 10% szybciej. Z drugiej strony, jeśli zupełnie usuniemy te 10%, program nasz będzie działał 10 razy szybciej Matematyk oczywiście przemawia za tym aby zająć się tymi 10%. Grupa OL twierdzi, że powinniśmy napisać kod zwracając tylko uwagę na wydajność (tj. dać wybór pomiędzy algorytmem $O(n^2)$ a O(n lg n), wybierając ten drugi). Ponieważ program działa poprawnie możemy wrócić i skoncentrować wysiłek na tych 10% kodu, który zabiera cały czas.

Argumenty OL są przekonywujące. Optymalizacja jest żmudnym i trudnym problemem. Najczęściej nie ma jasnego sposobu przyspieszenia sekcji kodu. Jedyny sposób określa, która z różnych opcji jest lepszym w rzeczywistości kodem i porównać je. Próba zrobienia tego na wejściu programu jest niepraktyczne. Jednakże, jeśli znajdziemy te 10% kodu i zoptymalizujemy go, zredukujemy nasz e obciążenie o 90% istotnie bardzo kusząc .Innym dobrym argumentem grupy)L jest to ,że niewielu programistów jest w stanie określić gdzie najwięcej czasu jest spędzanych w programie. Dlatego też jedynym rzeczywistym sposobem określenia gdzie program spędza czas jest oprzyrządowanie go i pomierzenie, które funkcje zabierają większość czasu. Oczywiście musimy mieć działający program zanim to zrobimy. Ponownie, argumentują, że czas spędzony nad

optymalizacją kodu z wyprzedzeniem jest marnotrawstwem ponieważ prawdopodobnie skończymy na optymalizacji tych 90%, które nie zmieniają niczego.

Są jednak bardzo dobre kontr argumenty do powyższych. Po pierwsze, kiedy większość OL zaczyna mówić o zasadzie 90/10, jest to bezgraniczna sugestia, że te 10% kodu pojawi się jako jeden duży fragment w środku naszego programu .Dobry programista, podobnie jak dobry chirurg, może zlokalizować tą złośliwą masę, wyciąć ją i zastąpić czymś dużo szybszym, a zatem zwiększyć szybkość programu niewielkim wysiłkiem. Niestety, nie jest to częsty przypadek w świecie rzeczywistym. W prawdziwych programach, te 10% kodu , które zabiera 90% czasu wykonania jest często porozrzucany po całym naszym programie. Będziemy mieli 1% tu, 0,5% tam, „gigantyczne" 2,5% w jednej funkcji i tak dalej. Gorzej jeszcze, optymalizacja 1% kodu wewnątrz jednej funkcji często wymaga aby zmodyfikować również jakiś inny kod. Na przykład, przepisując funkcję (1%), przyspieszając ją trochę, wymagana jest zmiana sposobu przekazywania parametrów do tej funkcji. Może to wymagać przepisania kilku sekcji kodu na zewnątrz, tych wolnych 10%. Wiec często kończy się na przepisaniu dużo więcej niż tylko 10% kodu aby przyspieszyć te 10 % zajmujące 90% czasu.

Inny problem z zasadą 90/10 jest taki, że dział na procentach, i zmienia procenty podczas optymalizacji. Na przykład przypuśćmy, że zlokalizowaliśmy funkcję pożerającą 90% czasu wykonania. Przypuśćmy, że Pan Super Programista i ty daliście sobie radę z przyspieszeniem tego podprogramu dwukrotnie. Nasz program pobierać bezie około 55% czasu wykonania przed optymalizacją. Jeśli potroimy szybkość podprogramu , program pobiera 40% całkowitego czasu wykonania. Jeśli spowodujemy ,że ta funkcja będzie działała dziewięć razy szybciej , program działał będzie teraz w 20% czasu oryginalnego, tj. pięć razy szybciej.

Przypuśćmy, że możemy pobrać funkcję działającą dziewięć razy szybciej. Załóżmy, że zasada 90/10 już nie ma zastosowania do naszego programu. 50% czasu wykonania jest spędzanych w 10% kodu, 50% w pozostałych 90% kodu. I jeśli poradziliśmy sobie z przyspieszeniem tej jednej funkcji 0 900%, jest bardzo nieprawdopodobne abyśmy musieli wykluczyć dużo więcej z niego. Czy jest warte zachodu nie traktowanie poważnie tych pozostałych 90% kodu? Zakładamy ,że jest. W końcu, możemy poprawić wydajność naszego programu o 25% jeśli podwoimy szybkość tego pozostałego kodu. Odnotujmy, że jednak mamy tylko 25% wydajność zwiększoną po optymalizacji 10%. Mając zoptymalizowane najpierw 90%, będziemy mieli 5% poprawę wydajności; prawie nic. Pomimo to ujrzymy sytuację gdzie zasada 90/10 wyraźnie nie ma zastosowania i zobaczymy przypadki gdzie optymalizacja tych 90% może stworzyć znaczną poprawę wydajności. Grupa OL uśmiechnie się i powie „hmm, to jest korzyść później optymalizacji, możemy optymalizować stopniowo i pobierać poprawną ilość optymalizacji jaką potrzebujemy".

Grupa wczesnej optymalizacji (OE) używa słabości w arytmetyce procentów do wskazania, że prawdopodobnie zakończymy optymalizację dużej części naszego programu tak czy inaczej. Więc dlaczego nie obsłużyć wszystkiego tego w pierwszej kolejności w naszym projekcie? Duży problem ze strategią OL jest taki, że często kończymy projektowanie i pisanie programu dwukrotnie – raz aby uczynić go funkcjonalnym, drugi raz czyniąc go praktycznym. W końcu jeśli mamy zamiar przepisać i tak 90%, dlaczego nie napisać go szybciej za pierwszym razem? Ludzie z OE również wskazują ,ze chociaż programiści notorycznie błądzą przy ustalaniu gzie program spędza większość czasu, są pewne oczywiste miejsca gdzie wiadomo iż wystąpią problemy z wydajnością. Dlaczego czekać do odkrycia tej oczywistości? Dlaczego nie obsłużyć takich obszarów problemu wcześniej co pozwoli spędzić mniej czasu na mierzeniu i optymalizacji tego kodu?

Podobnie jak wiele innych argumentów w Inżynierii Oprogramowania, te dwa obozy stały się zupełnie spolaryzowane i stają się zwolennikami całkowicie czystego podejścia w jednym z kierunków (albo OE albo OL). Podobnie jak wiele innych argumentów w Informatyce, prawda leży gdzieś pomiędzy tymi dwoma ekstremami Każdy projekt gdzie programista przystępuje do projektowania perfekcyjnego programu bez martwienia się o wydajność dopóki koniec jest przesądzony. .Większość programistów w tym scenariuszu pisze strasznie wolny kod. Dlaczego? Ponieważ łatwiej jest zrobić to a potem zawsze mogą „rozwiązać problem wydajności podczas fazy optymalizacji." W wyniku tego, 90% części programu jest często tak wolnych , że nawet jeśli czas pozostałych 10% zostałby zredukowany do zera program byłby jeszcze zbyt wolny. Z drugiej strony, grupa OE próbuje dogonić w pisaniu najlepszego z możliwych kodu opuszczając warunki graniczne , a produkt może nigdy nie zaskoczyć,

Jest jeden niezaprzeczalny fakt, który faworyzuje argumentację OL – kod optymalizowany jest trudny do zrozumienia i pielęgnacji. Co więcej, często zawiera błędy, które nie są obecne w kodzie niezoptymalizowanym. Ponieważ kod niepoprawny jest nieakceptowany, nawet jeśli działa szybciej, bardzo dobrym argumentem przeciw wczesnej optymalizacji jest fakt, ze testowanie, debuggowanie i zapewnienie jakości przedstawia dużą część cyklu rozwoju programu. Wczesna optymalizacja może tworzyć wiele dodatkowych błędów programowych, które gubimy obojętnie kiedy ,zachowując przez nie optymalizowanie później w cyklu rozwoju.

Poprawnym czasem do optymalizacji programu jest, cóż, odpowiedni czas. Niestety „odpowiedni czas" różni się w programie. Jednakże, pierwszym krokiem jest rozwój wydajności programu wymaganej wraz z innymi specyfikacjami programu. System analizy powinien rozwinąć docelowy czas reakcji dla wszystkich

interakcji użytkownika i przetwarzania. Podczas rozwoju i testowania, programiści maja cel do wypełnienia , więc nie mogą lenić się i czekać na fazę optymalizacji przed napisaniem kodu, wykonującego się rozsądnie. Z drugiej strony mają również na celowniku to, że chociaż kod działa dosyć szybko, nie muszą marnować czasu, lub czynić swój kod mniej pielęgnacyjnym; mogą pójść dalej i pracować nad resztą programu. Oczywiście, system analizy może źle ocenić wymagania wydajności, ale nie zdarza się to często w dobrze zaprojektowanym systemie.

Inną okolicznością jest to kiedy co wykonać. Jest kilka typów optymalizacji jakie możemy zastosować. Na przykład. Możemy przestawić instrukcje unikając hazardu dublujących szybkość kawałka kodu. Lub możemy wybrać różne algorytmy, które możemy uruchomi dwa razy szybciej. Jednym wielkim problem z optymalizacją jest taki, że nie jest to proces pojedynczy a wiele typów optymalizacji jest lepiej wykonać później niż wcześniej lub vice versa. Na przykład, wybór dobrego algorytmu jest tym co powinniśmy zrobić wcześniej. Jeśli zdecydujemy się użyć lepszego algorytmu po implementacji kiepskiego, wiele pracy na kodzie implementującym stary algorytm jest stracone. Podobnie szeregowanie instrukcji jest jedną z ostatnich optymalizacji jakie powinniśmy zrobić. Każda zmiana kodu po przestawieniu instrukcji dla wydajności, może wymusić spędzenie później wiele na czasu na przestawianiu ich ponownie. Najwyraźniej najniższy poziom optymalizacji (tj. zależny od CPU lub parametrów systemu) powinien być optymalizowany później. Odwrotnie, poziom najwyższy optymalizacji (tj. wybór algorytmu) powinien być optymalizowany szybciej. We wszystkich przypadkach powinniśmy mieć docelowe wartości wydajności na myśli podczas rozwijania kodu.

25.2 JAK ZNALEŻĆ WOLNY KOD W NASZYM PROGRAMIE?

Chociaż są problemy z zasada 90/10, koncepcja jest w zasadzie solidna – programy mają w zwyczaju spędzać duża ilość swojego czasu wykonania tylko w małym procencie kodu. Wyraźnie powinniśmy zoptymalizować najpierw najwolniejszą część kodu. Jedyny problem to ten jak znaleźć ten najwolniejszy kod programu?

Są cztery powszechne techniki programistyczne używane do znajdowania „gorących miejsc" (miejsc gdzie programy spędzają większość swojego czasu). Pierwsza to metoda prób i błędów. Druga to optymalizacja wszystkiego. Trzecią jest analiza programu. Czwartą jest użycie profilu lub innego systemowego narzędzia monitorowania wydajności różnych części programu. Po zlokalizowaniu gorących miejsc programista może próbować zanalizować tą część programu.

Technika prób i błędów jest ,niestety, najpopularniejszą strategią. Programista przyspiesza różne części programu poprzez uczynienie świadomego odgadywania gdzie spędza się większość czasu. Jeśli programista odgadnie prawidłowo, program będzie działał dużo szybciej po optymalizacji. Eksperymentujący programiści często używają tej techniki szczęśliwie szybko lokalizując i optymalizując program. Jeśli programista odgadnie prawidłowo, technika ta minimalizuje ilość czasu spędzonego na znajdowaniu gorących miejsc w programie. Niestety większość programistów źle odgaduje i kończy na optymalizowaniu złych części kodów. Taki wysiłek często idzie na marne ponieważ optymalizacja złych 10% nie poprawi zdecydowanie wydajności. Jednym z podstawowych powodów błędności tej techniki jest to ,że jest często pierwszym wyborem niedoświadczonych programistów , którzy nie mogą łatwo rozpoznać wolnego kodu. Niestety, są oni nieświadomi innych technik, więc zamiast próbować podejścia strukturalnego, zaczynają stosować (często) odgadywanie nieświadome.

Innym sposobem zlokalizowania i optymalizacji wolnej części programu jest optymalizacja wszystkiego. Oczywiście technika ta nie działa dobrze dla dużych programów, ale dla krótkich części kodu działa stosunkowo dobrze. Później w tym tekście będzie dostarczony krótkiego przykładu problemu optymalizacji i będzie stosował tą technikę optymalizacji programu. Oczywiście, dla dużego programu lub podprogramów może nie być podejściem mało opłacalnym. Jednakże gdzie właściwie można zachować czas podczas optymalizacji naszego programu (lub przynajmniej części programu) ponieważ nie będziemy musieli uważnie analizować i mierzyć wydajności naszego kodu. Poprzez optymalizację wszystkiego, jesteśmy pewni optymalizacji wolnego kodu.

Metoda analizy jest najtrudniejszą z tych czterech. Przy tej metodzie studiujemy nasz kod i określamy gdzie program spędza większość czasu w oparciu o dane jakich oczekujemy od procesu. Teoretycznie , jest to najlepsza technika. W praktyce, istoty ludzkie generalnie wykazują dystans dla takiej pracy analitycznej. Jako taka, analiza jest często niepoprawna lub zbyt długa do ukończenia. Co więcej, niewielu programistów ma duże doświadczenie w studiowaniu swoich kodów dla określenia gdzie spędza on większość swojego czasu, więc często są bezradni przy lokalizowaniu gorących miejsc przez studiowanie swoich listingów kiedy pojawia się potrzeba.

Pomimo problemów z analizą programów, jest to pierwsza technika jakiej powinniśmy zawsze używać, kiedy próbujemy optymalizować program.. Prawie wszystkie programy spędzają większość swojego czasu wykonania w ciele pętli lub rekurencyjnym wywołaniu funkcji.. Dlatego też powinniśmy spróbować zlokalizować wszystkie rekurencyjne wywołania funkcji (zwłaszcza pętle zagnieżdżone) w naszych

programach. Jest bardzo duże prawdopodobieństwo, że program będzie spędzał większość swojego czasu w jednym z tych dwóch obszarach programu. Takie miejsca są do rozważenia w pierwszej kolejności kiedy optymalizujemy nasze programy.

Chociaż metoda analityczna dostarcza dobrego sposobu zlokalizowania wolnego kodu w programie, analizowanie programu jest wolnym, nużącym i nudnym procesem. Jest bardzo łatwo zupełnie zgubić najbardziej czasochłonna część programu, zwłaszcza w pośredniej obecności wywołania funkcji rekurencyjnej. Nawet zlokalizowanie czasochłonnych pętli zagnieżdżonych jest często trudne. Na przykład możemy nie uświadamiać sobie, kiedy szukamy pętli wewnątrz procedury, że jest zagnieżdżona pętla z racji faktu, że kod wywołujący wykonuje pętlę kiedy wywołujemy procedurę. Teoretycznie, metoda analityczna powinna zawsze działać. W praktyce, jest to tylko nieznaczny sukces ,kiedy omylni ludzie robią analizę. Niemniej jednak, pewne gorące miejsca są łatwe do odnalezienia poprzez analizę programu, więc naszym pierwszym krokiem, kiedy optymalizujemy program jest analiza.

Ponieważ programiści są notorycznie słabi przy analizie programów w celu znalezienia gorących miejsc, można spróbować zautomatyzować ten proces. Jest to dokładnie to co profiler może zrobić dla nas. Profiler jest małym programikiem, który mierzy jak długo nasz kod spędza w jakiejś części programu. Profiler zazwyczaj działa poprzez cykliczne przerywanie naszego kodu i odnotowywanie adresu powrotnego. Profiler buduje histogram przerwań adresów przerwań (generalnie zaokrągla do określonej przez użytkownika wartości). Poprzez studiowanie tego , możemy określić gdzie program spędza większość czasu .Powie to nam jaką część kodu musimy zoptymalizować. Oczywiście, stosując ta technikę, będziemy potrzebowali programu profilera. Borland, Microsoft i kilku innych producentów dostarcza profilerów i innych narzędzi optymalizujących.

25.3 CZY OPTYMALIZACJA JST KONIECZNA?

Z wyjątkiem zabawy i edukacji, nigdy nie powinniśmy próbować podejścia do projektu z postawą, że musimy uzyskać maksymalną wydajność naszego kodu. Lata temu, była to ważna postawa ponieważ było to to co pozwalało uzyskiwać przyzwoite działanie na wolnych maszynach tej ery. Redukując czas działania programu z dziesięciu minut do dziesięciu sekund, uczyniono poważny krok dla poprawy wydajności . Z drugiej strony przyspieszenie programu pobierającego 0,1 sekundy do punktu w którym działa on w milisekundę jest często bezcelowe. Zmarnujemy dużo wysiłku poprawiając wydajność, mimo to tylko kilka osób odnotuje różnice.

Nie mówię, że przyspieszenie programu z 0,1 sekundy do 0,001 sekundy nie jest nigdy warte zachodu. Jeśli piszemy program do zbierania danych, który wymaga wykonania odczytu co milisekundę, a może tylko obsłużyć dziesięć odczytów na sekundę jako obecnie zapisanych. Co więcej, nawet jeśli nasz program działu już dość szybko, są powody dlaczego chcielibyśmy uczynić go dwa razy szybszym. Na przykład przypuśćmy, że ktoś może używać naszego programu w środowisku wielozadaniowym. Jeśli zmodyfikujemy nasz program do dwukrotnie szybszego, użytkownik będzie mógł uruchomić inny program wraz z naszym i nie zauważyć obniżenia wydajności.

Jednakże, sprawą do zapamiętania jest to, że musimy napisać oprogramowanie, które jest dosyć szybkie. Ponieważ program tworzy wyniki natychmiastowe (lub prawie bliskie natychmiastowych) istnieje potrzeba uczynienie jego uruchomienia szybszym. Ponieważ optymalizacja jest procesem kosztownym i skłonnym do błędów, chcemy unikać jej jak to tylko możliwe. Pisanie programów, które działają szybciej niż dosyć szybki jest marnowaniem czasu. Jednakże, jak widać wyraźnie z dzisiejszych rozdętych aplikacji, nie jest to rzeczywisty, większość tworzonych kodów programistycznych jest zbyt wolnych, nie zbyt szybkich.

Popularnym podawanym powodem dla tworzenia nie zoptymalizowanego kodu jest złożoność sprzętu. Wielu programistów i menadżerów czuje ze maszyny wyższej klasy dla których projektują oprogramowanie dzisiaj, będą maszynami średniej klasy za dwa lata, kiedy udostępnią końcową wersją oprogramowania .Więc jeśli zaprojektują oprogramowanie działające na dzisiejszych bardzo szybkich maszynach, będą się wykonywały średniej klasy maszynach kiedy udostępnią oprogramowanie

Są z tym związane dwa problemy. Po pierwsze system operacyjny działający na tych maszynach za dwa lata pożre znaczną część zasobów maszyny (wliczając w to cykle CPU) Ciekawe jest, że dzisiejsze są sto razy szybsze niż oryginalny 8088, a mimo to wiele aplikacji działa wolniej niż gdyby działał na oryginalnym PC.. Prawda , dzisiejsze oprogramowanie dostarcza wiele cech poza te z oryginalnego PC, ale jest cała masa argumentów – klienci domagają się cech takich jak wiele okienek, GUI, menu rozwijane itd., które wykorzystują cykle CPU. Nie możemy zakładać, ze nowsze maszyny będą dostarczały dodatkowych cykli zegarowych, więc nasz wolny kod będzie działał szybciej. OS lub interfejs użytkownika naszego programu zakończy zjedzenie tych dodatkowych dostępnych cykli zegarowych.

Tak więc pierwszym krokiem jest realistyczne określenie żądanej wydajności naszego programu. Potem musimy napisać program spełniający ten cel wydajności. Kiedy rozminiemy się z żądana wydajnością, wtedy

jest czas na optymalizację programu. Jednakże nie powinniśmy marnować dodatkowego czasu optymalizując kod ponieważ nasz program napotyka lub przekracza specyfikację wydajności.

25.4 TRZY TYPY OPTYMALZACJI

Są trzy formy optymalizacji jakie może zastosować, kiedy poprawiamy wydajność programu. Wybierają one lepszy algorytm (optymalizacja wysokiego poziomu), implementację lepszego algorytmu (poziom optymalizacji średniego poziomu) i „zliczanie cykli" (optymalizacja niskiego poziomu). Każda technika ma swoje miejsce i, generalnie, stosujemy je w różnych miejscach w procesie projektowania.

Wybór lepszego algorytmu jest najbardziej nagłośnioną technika optymalizacji. Niestety jest to technika używana najmniej często. Jest łatwa dla kogoś głoszącego, że powinniśmy zawsze znajdować lepszy algorytm jeśli potrzebujemy większej szybkości; ale znalezienie tego algorytmu jest trochę trudniejsze. Po pierwsze, zdefiniujmy algorytm zmiany ponieważ używamy zasadniczo różnych technik do rozwiązania problemu. Na przykład, przełączając z algorytmu „sortowania bąbelkowego" do algorytmu „szybkiego sortowania" jest dobrym przykładem algorytmu zmiany, Generalnie, chociaż nie zawsze, zmiany algorytmów oznaczają, że używamy programu z lepsza funkcją Big-Oh. Na przykład, kiedy przełączamy z sortowania bąbelkowego do szybkiego sortowania, zmieniamy algorytm z czasem działania $O(n^2)$ na algorytm z oczekiwanym czasem działania O(n lg n).

Musimy pamiętać o ograniczeniach funkcji Big-Oh kiedy porównujemy algorytmy. Wartość dla n musi być wystarczająco duża do zamaskowania efektu ukrytej stałej. Co więcej, analizy Big-Oh są zazwyczaj najgorsze i mogą nie wpływać na nasz program. Na przykład jeśli życzymy sobie posortować tablicę, która jest „prawie" posortowana najpierw, algorytm sortowania bąbelkowego jest zazwyczaj szybszy niż algorytm szybkiego sortowania, bez względu na wartość dla n. Dla danej, która jest prawie posortowana, sortowanie bąbelkowego działa prawie w czasie O(n) podczas gdy algorytm szybkiego sortowania działa w czasie $O(n^2)$.

Drugą rzeczą do zapamiętania jest stałość. Jeśli dwa algorytmy mają taką samą funkcję Big-Oh, nie możemy określić żadnej różnicy pomiędzy dwoma analizami Big-Oh. To nie znaczy, że będą one pobierały taką samą ilość czasu działania. Nie zapomnijmy, że w analizie Big-Oh odrzucamy wszystkie nisko poziomowe warunki i stałe mnożne. Trochę bardziej pomocny jest zapis asymptotyczny w tym przypadku.

Uzyskanie rzeczywiście poprawy wydajności wymaga zmiany algorytmu naszego programu. Jednakże, zmiana algorytmu O(n lg n) na algorytm $O(n^2)$ jest często trudniejsze jeśli rozwiązanie już nie istnieje. Przypuszczalnie , dobrze zaprojektowany program nie będzie zawierał oczywistych algorytmów, które możemy dramatycznie poprawić (jeśli są , nie były dobrze zaprojektowane).Dlatego też, próby znalezienia lepszego algorytmu może nie zakończyć się powodzeniem. Niemniej jednak jest to zawsze pierwszy krok jaki powinniśmy wykonać ponieważ kolejne kroki działają na takim algorytmie jaki mamy. Jeśli wykonujemy te inne kroki na złym algorytmie a potem, później odkryjemy lepszy algorytm, będziemy musieli powtarzać, te czasochłonne kroki ponownie przy nowym algorytmie.

Są dwa kroki odkrywania nowych algorytmów: badanie i rozwój. Pierwszy krok służy temu aby zobaczyć czy możemy znaleźć lepsze rozwiązanie w istniejącej literaturze. Ewentualnie, drugi krok jest po to aby zobaczyć czy możemy odkryć lepszy algorytm w naszym własnym. Kluczem do tego jest zabudżetowanie właściwej ilości czasu dla tych dwóch działań badanie jest luźnym procesem . Zawsze możemy przeczytać więcej książek lub artykułów. Więc musimy zadecydować jak wiele czasu mamy zamiar spędzić szukając istniejącego rozwiązania. Może to być kilka godzin, dni m, tygodni lub miesięcy. Jakkolwiek jest to opłacalne. Potem możemy udać się do biblioteki (lub półki z książkami) i poszukać lepszego rozwiązania. Gdy tylko wygasa nasz czas, czas porzucić podejście badawcze chyba, że jesteśmy pewni, że zmierzamy we właściwym kierunku w materiałach jakie studiujemy. Jeśli tak zabudżetujemy trochę więcej czasu i zobaczmy jak to działa. W tym samym miejscu jednak musimy zadecydować czy prawdopodobnie nie można znaleźć lepszego rozwiązania i czy jest czas aby rozwinąć nowe w naszym własnym.

Podczas poszukiwania lepszego rozwiązania powinniśmy studiować dokładnie artykuły, teksty itp. Chociaż przestudiowaliśmy ważne testy. Kiedy prawdą jest ,że większość z tego co wystudiowaliśmy nie będzie miało zastosowania do problemu, nauczymy się o rzeczach, które będą użyteczne w przyszłych projektach. Co więcej, podczas gdy ktoś może nie uzyskać potrzebnego rozwiązania, może wykonać pracę , która idzie w tym samym kierunku co nasza i może dostarczyć takich samych dobrych pomysłów, jeśli nie podstawowych dla naszego własnego rozwiązania. Jednakże, zawsze musimy pamiętać, że zadaniem inżynierów jest dostarczenie wydajnych rozwiązań problemu. Jeśli marnujemy zbyt dużo czasu szukając rozwiązań, które mogą się nigdzie nie pojawić w literaturze, spowodujemy kosztowne przedłużenie naszego projektu. Trzeba wiedzieć, kiedy jest czas aby „odłożyć go" i zająć się resztą projektu.

Rozwijanie nowego algorytmu jako naszego własnego jest również luźne. Możemy spędzić resztę życia próbując znaleźć wydajne rozwiązanie dla trudnego do rozwiązania problemu. Ponownie więc musimy zabudżetować czas w związku z tym dla tego projektu. Spędzajmy czas mądrze próbując rozwijać lepsze

rozwiązanie dla naszego problemu ,ale ponieważ czas jest nieubłagany, czas spróbować różnych podejść zamiast marnować czas na ściganie „świętego grala".

Bądźmy pewni użycia wszystkich zasobów będących w naszej dyspozycji, kiedy próbujemy znaleźć lepszy algorytm. Lokalna biblioteka uniwersytecka może być bardzo pomocna. Również powinniśmy skorzystać z Sieci. Uczęszczając na spotkania miejscowego klubu komputerowego, omawiając nasz problem z innymi, lub rozmawiając z przyjaciółmi, być może ktoś czytał o rozwiązaniu tego czego szukamy. Jeśli mamy dostęp do Internetu, BIX, Compuserv lub innych serwisów on-line bądź BBS'ów, naturalnie możemy wysłać post z informacją o pomoc. Wśród milionów użytkowników, jeśli istnieje lepsze rozwiązanie naszego problemu, ktoś prawdopodobnie już go rozwiązał. Kilka postów może zwiększyć rozwiązanie, którego nie byliśmy w stanie znaleźć lub rozwiązać sami.

W tym miejscu musimy przyznać się do niepowodzenia. W rzeczywistości możemy musieć przyznać się do powodzenia – już znaleźliśmy tak dobry algorytm jak można było. Jeśli ten jest jeszcze zbyt wolny dla naszych wymagań, może być czas aby spróbować innych technik poprawy szybkości programu. Kolejnym krokiem jest sprawdzenie czy możemy dostarczyć lepszej implementacji dla algorytmu jakiego używamy. Ten krok optymalizacji, chociaż niezależny od języka, właśnie dla większości programistów asemblerowych tworzy zdecydowaną poprawę wydajności ich kodu. Lepsza implementacja generalnie wymaga kroków takich jak pętle rozwijane, używając przeszukiwania tablicy zamiast obliczeń, eliminując obliczenia z pętli, której wartość nie zmienia się wewnątrz pętli, wykorzystując idiomy maszynowe(tak jak użycie shift lub shift i and zamiast mnożenia), próbując zachować zmienne w rejestrach tak długo jak to możliwe i tak dalej. Jest niespodzianką o ile szybciej program może działać poprzez zastosowanie prostych technik jak te, których opisy pojawiły się w tym tekście.

W ostateczności możemy się uciec do zliczania cykli.. Na tym poziomie możemy próbować założyć, że sekwencja instrukcji używa kilku cykli zegarowych. Jest to trudne do optymalizacji do wykonania ponieważ musimy być świadomi jak wiele cykli zegarowych konsumuje każda instrukcja, a to zależy od instrukcji, używanego trybu adresowania, instrukcji dookoła instrukcji bieżącej (tzn. efekty potokowy i superskalarny), szybkości systemu pamięci (stany oczekiwania i pamięci podręcznej) i tak dalej. Rzecz jasna, takie optymalizacje są bardzo nużące i wymagają bardzo ostrożnej analizy programu i systemu na którym działamy.

Zwolennicy OL zawsze domagają się aby odłożyć optymalizację na tak długo jak to możliwe .Ludzie ci generalnie mówią o tym jako szybkiej formie optymalizacji. Powód jest prosty: każda zmiana jakiej dokonamy po takiej optymalizacji może zmienić oddziaływanie instrukcji i ich czas wykonania. Jeśli spędzamy znaczna część czasu wybierając sekwencję 50 instrukcji a potem odkrywamy ,ze musimy przepisać ten kod z powodu tego czy innego, cały ten czas spędzamy starannie wybierając te instrukcje unikając hazardu . Z drugiej strony, jeśli czekamy na ostatni możliwy moment do zrobienia takiej optymalizacji naszego kodu, zoptymalizujemy tylko taki kod.

Wielu programistów HLL powie, że dobry kompilator może pokonać człowieka przy wybieraniu instrukcji i optymalizacji kodu. To nie jest prawda. Dobry kompilator pokona pośledni program asemblerowy w dużej mierze. Jednakże, dobry kompilator nie starcza na dobrego programistę asemblerowego. W końcu, najgorsze co się może zdarzyć jest to, że dobry programista asemblerowy będzie patrzył na wyjście kompilatora i poprawiał go.

„Zliczanie cykli" może poprawić wydajność naszego programu Średnio, możemy przyspieszyć program od 50 do 200% wykonując proste zmiany (jak przestawienie instrukcji). To jest różnica pomiędzy 80486 a Pentium! Więc nie powinniśmy ignorować możliwości zastosowania takiej optymalizacji naszych programów. Zapamiętajmy, że powinniśmy robić taką optymalizację wcześniej zanim nie zakończymy przerabiać ich ponieważ zmienił się nasz kod.

Reszta tego rozdziału skoncentruje się na tych technikach poprawy implementacji algorytmu , zamiast projektować lepszy algorytm lub stosować techniki zliczania cykli. Projektowanie lepszego algorytmu wykracza poza temat tego podręcznika. Zliczanie cykli jest jednym z tych procesów, który różni się w zależności od procesora. To znaczy, techniki optymalizacji które działają dobrze na 80386 zawodzą na 486 lub Pentium i vice versa. Ponieważ Intel stale tworzy nowe chipy, wymagające różnych technik optymalizacjnych, listując te techniki tu, materiał ten może okazać się przestarzały .Intel publikuje takie podpowiedzi optymalizacyjne w swoich podręcznikach. Artykuły na temat optymalizacji programów asemblerowych często pojawiają się w magazynach technicznych takich jak Dr. Dobb's Journal, powinniśmy czytać takie artykuły i uczyć wszystkich technik optymalizacyjnych.

25.5 POPRAWA IMPLEMNTCJI ALGORYTMU

Jedynym prostym sposobem częściowego zademonstrowania jak zoptymalizować kawałek kodu jest dostarczenie jakiegoś przykładu i kroków optymalizacyjnych jakie możemy zastosować do tego programu. Sekcja ta przedstawi krótki program, który rozmywa obrazek w ośmiobitowej skali szarości. Potem sekcja ta

poprowadzi prze kilka kroków optymalizacyjnych i pokaże jak uczynić program działającym 16 razy szybciej. Poniższy kod zakłada, że dostarczamy go z plikiem zawierającym zdjęcie w skali szarości 251x256. Struktura danych dla tego pliku jest następująca:

```
Image: array [0..255, 0..255] of byte
```

Każdy bajt zawiera wartości w zakresie 0..255 z zerem oznaczającym czerń, 255 przedstawiającym biel a pozostałe wartości przedstawiają odcienie szarości pomiędzy tymi dwoma wartościami ekstremalnymi.
Algorrytm rozmycia uśrednia piksel wraz z jego ośmioma najbliższymi sąsiadami. Pojedyncza operacja rozmycia stosuje tą średnią dla wszystkich wewnętrznych pikseli obrazka (to znaczy, nie stosuje pikseli na granicy obrazka ponieważ nie ma takiej samej liczby sąsiadujących pikseli jak inne piksele) poniższy program Pascalowski implementuje algorytm rozmycia i pozwala użytkownikowi określić ilość rozmyć (poprzez zapętlenie w całym algorytmie ilości określonych przez użytkownika):

```
Program PhotoFilter(input, output);

(* Tu jest plik z danymi pierwotnymi stworzony przez program Photoshop *)
type
        image = array [0..255] of array [0..255] of byte

(* Zmienne jakich będziemy używać. Zauważmy, że  zmienne „datain" i dataout" są wskaźnikami ponieważ *)
(* Turbo Pascal nie pozwoli nam zaalokować więcej niż 64K danych w jednym globalnym segmencie danych*)

var

h ,i, j ,k ,l, sum ,iterations : integer;
datain, dataout : ^image;
f, g: file of image;

begin

(* Otwieramy pliki I rzeczywistą daną wejściową*)

assign (f, 'roller1.raw');
assign (f,roller2.raw');
reset(f);
rewrite(g);
new (datain);
new (dataout);
read(f, datain^);

(* Pobranie liczby iteracji od użytkownika *)

write ('Wprowadź liczbę iteracji: ')
readln(iterations);

writeln (Obliczanie wyniku');

(* Kopiowanie danych z tablicy wejściowej do tablicy wyjściowej. W rzeczywistości jest to kiepski sposób *)
(* kopiowania z tablicy wejściowej do tablicy wyjściowej *)

for i:=0 to 255 do
        for j:=0 to 255 do
                dataout^ [i][j] := datain^ [i][j];

(* Okay, jesteśmy tu gdzie wykonuje się cała praca. Pętla zewnętrzna powtarza operację rozmywania ilość *)
(* razy określoną przez użytkownika                                                                    *)

for h:=1 to iterations do begin
```

```
        (*Dla każdego wiersza pierwotnej z wyjątkiem pierwszej i ostatniej, obliczamy nową wartość*)
        (*dla każdego elementu                                                              *)

                for i:= 1 to 249 do
        (* Dla każdej kolumny z wyjątkiem pierwszej i ostatniej obliczamy nową wartość dla każdego*)
        (* elementu                                                                          *)

        for j:= 1 to 254 do begin
                (*Dla każdego elementu w tablicy, obliczamy nową wartość rozmycia poprzez dodanie ośmiu
                  komórek wokół elementu tablicy osiem razy do bieżącej wartości komórki. Potem dzielimy
                  to przez szesnaście obliczając średnią z dziewięciu komórek tworzących kwadrat wokół
                  bieżącej komórki. Bieżąca komórka ma 50% znaczenie, pozostałe osiem komórek wokół
                  bieżącej komórki dostarcza pozostałych 50% (6,25% każda) *)

                sum := 0
                        for k := -1 to 1 do
                                for l:= -1 to 1 do
                                        sum :=sum+datain^[I+k] [j+l];
                (* Sum aktualnie zawiera sumę dziewięciu komórek, dodanych siedem razy do bieżącej   *)
                (* komórki, więc uzyskujemy całkowicie ośmiokrotnie bieżącą komórkę                  *)

                dataout^ [i][j] := (sum +datain^ [i][j]*7) div 16;

end;

(* Kopiowanie wartości komórki wyjściowej z powrotem do komórki wejściowej, wiec możemy osiągnąć  *)
(* rozmycie z tą nową daną w kolejnej iteracji                                                     *)

for i:= 0 to 250 do
  for j:= 0 to 255 do
        datain^ [i][j] := dataout^ [i][j];

end;

writeln ('Zapis wyników');
write(g, dataout^);
close(f);
close(g);
end.
```

Powyższy program Pascala skompilowany z Turbo Pascalem v 7.0 , w 45 sekund oblicza 100 iteracji algorytmu rozmycia. Porównywalny program napisany w C i skompilowany pod Borland C++ v 4.02 zajmuje 29 sekund do działania .taki sam plik źródłowy skompilowany z Microsoft C++ v 8.00 działa w 21 sekund. Oczywiście, kompilatory C tworzą lepszy kod niż Turbo Pascal.. Zajęło około 3 godzin uzyskanie wersji pascalowskiej działającej i przetestowanej. Wersja C zajęła około jednej godziny na zakodowanie i przetestowanie. Poniższe dwa obrazki to przykłady „przed" i „po" tej funkcji programu"

Przed rozmyciem:

Po rozmyciu (10 iteracji):

Poniżej mamy surowe tłumaczenie z Pascala bezpośrednio do języka asemblera powyższego programu. Wymaga 36 do działania. Tak, kompilatory C wykonują tę pracę lepiej, ale ponieważ widać jak zły to kod, będziemy zdumieni jak wolno działa kod Turbo Pascala. Zajmie około godziny przetłumaczenie wersji

Pascalowej na kod asemblerowy i zdebuggowanie go do miejsca w którym stworzymy takie same dane wyjściowe jak wersja Pascalowa.

```
; IMGPRCS.ASM
;
; Program przetwarzający obrazek
;
; Jest to program rozmywający obrazek w ośmiobitowej skali szarości poprzez uśrednienie piksela w obrazku z
; ośmioma pikselami wokół. Średnia jest obliczana przez (CurCell*8+ pozostałe 8 komórek)/16,  obciążając
; bieżącą komórkę o 50%
;
; Ponieważ rozmiar obrazka (prawie 64k), macierze wejściowe i wyjściowe są w różnych segmentach.
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
;
; Porównanie osiągów (system 66MHz 80486 DX/2)
;
; Ten kod                       36 sekund
; Borland Pascal v7.0           45 sekund
; Borland C++ v4.02             29 sekund
; Microsoft C++ v8.00           21 sekund

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list
                .286

dseg            segment para  public 'data'

;Pętla sterująca zmiennych i inne zmienne:

h               word    ?
i               word    ?
j               word    ?
k               word    ?
l               word    ?
sum             word    ?
iterations      word    ?

; Nazwa pliku:

InName          byte    "roller.raw", 0
OutName         byte    "roller2.raw",0

dseg            ends

; Tu mamy dane wejściowe na których działamy

InSeg           segment para public 'indata'
DataIn          byte    251  dup (256 dup (?))
InSeg           ends

; Tu jest tablica wyjściowa, która przechowuje wynik

OutSeg          segment para public 'outdata'
DataOut         byte    251 dup (256 dup(?))
OutSeg          ends

cseg            segment para public 'code'
```

```
                assume  cs:cseg, ds: dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                meminit

                mov     ax, 3d00h                       ;otwarcie pliku wejściowego do odczytu
                lea     dx, InName
                int     21h
                jnc     GoodOpen
                print
                byte    „Nie można otworzyć pliku .”,cr, lf,0
                jmp     Quit

GoodOpen:       mov     bx, ax                          ;uchwyt pliku
                mov     dx, InSeg                       ;gdzie pobieramy dane
                mov     d, dx
                lea     dx, DataIn
                mov     cx, 256*251                     ;rozmiar pliku danych do odczytu
                mov     ah, 3Fh
                int     21h
                cmp     ax, 256*251                     ;zobaczmy czy odczytamy daną
                je      GoodRead
                print
                byte    „Nie odczytano właściwie pliku”, cr, lf, 0
                jmp     Quit

GoodRead:       mov     ax, dseg
                mov     ds, ax
                print
                byte    “Wprowadź liczbę iteracji: “, 0
                getsm
                atoi
                free
                mov     iterations, ax
                print
                byte    „Obliczanie wyniku”, cr, lf, 0

; Kopiowanie danej wejściowej do bufora wyjściowego

                mov     i, 0
illop0:         cmp     i, 250
                ja      iDone0
                mov     j, 0
jloop0:         cmp     j, 255
                ja      jDone0

                mov     bx, I                   ;obliczenie indeksu  do obu tablic
                shl     bx, 8                   ;używając formuły i*256+j
                add     bx, j
                mov     cx, InSeg               ;wskazuje segment wejściowy
                mov     es, cx
                mov     al., es :DataIn[bx]             ;Pobranie DataIn[i][j]

                mov     cx, OutSeg                      ; wskazuje segment wyjściowy
                mov     es, cx
                mov     es: DataOut[bx], al.            ;Przechowanie w DataOut [i][j]
```

```
                inc     j                       ;kolejna iteracja pętli j
                jmp     jloop0
jDone0:         inc     I                       ;kolejna iteracja pętli I
                jmp     iloop0

iDone0:

; for h :=1 to iterations-
                mov     h, 1
hloop:          mov     ah, h
                cmp     ax, iterations
                ja      hloopDone

; for I:=1 to 249 –

                mov     I, 1
iloop:          cmp     1, 249
                ja      iloopDone

;for j :=1 to 254 –

                mov     j, 1
jloop:          cmp     j, 254
                ja      jloopDone

;sum :=0;
; for k := -1 to 1 do for l :=-1 to 1 do

                mov     ax, InSeg               ;uzyskanie dostępu do InSeg
                mov     es, ax
                mov     sum 0
                mov     k, -1
kloop:          cmp     k, 1
                jg      kloopDone

                mov     l, -1
lloop:          cmp     l, 1
                jg      lloopDone

;sum :=sum+datain [I+k][j+l]

                mov     bx, I
                add     bx, k
                shl     bx, 8                   ;mnożenie przez 256
                add     bx, j
                add     bx ,l

                mov     al., es: DataIn[bx]
                mov     ah, 0
                add     Sum, ax

                inc     l
                jmp     lloop
lloopDone:      inc     k
                jmp     kloop

;dataout[i][j] := (sum + datain[i][j] * 7) div 16;

kloopDone:      mov     bx, 1
```

```
                shl     bx, 8                   ;* 256
                add     bx, j
                mov     al, es:DataIn[bx]
                mov     ah, 0
                imul    ax, 7
                add     ax, sum
                shr     ax, 4                   ;div 16

                mov     bx ,OutSeg
                mov     es, bx
                mov     bx, i
                shl     bx, 8
                add     bx, j
                mov     es: DataOut[bx], al

                inc     j
                jmp     jloop
jloopDone:      inc     I
                jmp     iloop
iloopDone:

; Kopiowanie danej wyjściowej do bufora wejściowego

                mov     i, 0
iloop:          cmp     i, 250
                ja      iDone1
                mov     j, 0
                cmp     j, 255
                ja      jDone1

                mov     bx, i                   ;obliczanie indeksu do obu tablic
                shl     bx, 8                   ;używając formuły i*256+j
                add     bx, j

                mov     cx, OutSeg              ;wskazuje segment wejściowy
                mov     es, cx
                mov     al., es: DataOut[bx]    ; Pobranie DataIn[i][j]

                mov     cx, InSeg               ;wskazuje segment wyjściowy
                mov     es, cx
                mov     es: DataIn[bx], al.     ;przechowanie  DataOut[i][j]

                inc     j                       ;kolejna iteracja pętli j
                jmp     jloop1
jDone1:         inc     I                       ;kolejna iteracja pętli I
                jmp     iloop1
iDone1:         inc     h
                jmp     hloop
hloopDone:      print
                byte    “Zapisanie wyniku”, cr,lf,0

; Okay, zapisujemy daną do pliku wyjściowego:

                mov     ah, 3ch                 ;tworzenie pliku wyjściowego
                mov     cx, 0                   ;normalne atrybuty pliku
                lea     dx, OutName
                int     21h
                jnc     GoodCreate
                print
```

```
                byte    „Nie można stworzyć pliku wyjściowego”, cr, lf,0
                jmp     Quit
GoodCreate:     mov     bx, ax                  ;uchwyt pliku
                push    bx
                mov     dx, OutSeg              ;gdzie może być znaleziona dana
                mov     ds., dx
                lea     dx, DataOut
                mov     cx, 256*251             ;rozmiar danej pliku do zapisu
                mov     ah, 40h                 ;operacja zapisu
                int     21h
                pop     bx                      ;odzyskanie uchwyty dla zamknięcia
                cmp     ax, 256*251             ;zobaczmy czy zapisano daną
                je      GoodWrite
                print
                byte    „Nie zapisano poprawnie pliku”, cr, lf, 0
                jmp     Quit

GoodWrite:      mov     ah, 3eh             ;operacja zamknięcia
                Int     21h
Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment para stack ‘stack’
stk             byte    1024 dup (“stack”)
sseg            ends
zzzzzzseg       segment para public ‚zzzzzz’
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

Ten kod asemblerowy jest bardzo prosty, linia po linii tłumaczy poprzedni kod Pascalowski. Nawet początkujący programista (który przeczytał i zrozumiał Rozdziały Osiem i Dziewięć) powinien łatwo poprawić osiągi tego kodu.

Podczas kiedy możemy uruchomić profiler w tym programie określamy gdzie w tym kodzie są „gorące miejsca” , trochę analizy, zwłaszcza w wersji Pascalowej powinno uczynić oczywistym, że jest kilka zagnieżdżonych pętli w tym kodzie .jak wskazuje Rozdział dziesiąty, kiedy optymalizujemy kod powinniśmy zawsze zaczynać od pętli najskrytszych. Ważną zmianą miedzy powyższym kodem a wersją asemblerową jest to, że rozwijamy pętle najskrytsze i zamieniamy obliczony indeks tablicy z jakimś stałym obliczeniem. Ta drobna zmiana przyspiesza wykonywanie sześciokrotnie! Wersja asemblerowa działa teraz sześć sekund zamiast 36. Wersja Microsoft C++ tego samego programu z porównywalna optymalizacja działa osiem sekund. Wymaga blisko czterech godzin wywołanie, testowanie i zdebuggowanie tego kodu. Wymaga dodatkowej godziny zastosowanie tej samej modyfikacji co wersja C

```
; IMGPRCS2.ASM
;
; Program przetwarzający obrazek
;
; Program ten rozmywa obrazek w ośmiobitowej skali szarości poprzez piksela w tym obrazku z ośmioma
; pikselami wokół. Średnia jest obliczana z (CurCell*8 + pozostałe 8 komórek)/ 16 , obciążając aktualną
;  komórkę o 50%.
;
; Ponieważ rozmiar obrazka to prawie 64 K, matryce wejściowa i wyjściowa są w różnych segmentach.
;
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
; Wersja #2: Trzy główne optymalizacje. (1) używa instrukcji movsd zamiast pętli do kopiowania danych z
; DataOut z powrotem do DataIn. (2) Używa formy repeat...until dla wszystkich pętli (3) rozwija najskrytsze
; dwie pętle (które są odpowiedzialne za większość poprawiania wydajności)
;
;       Porównania wydajności (system 66 MHz 80486 DX/2)
```

```
;
;       Ten kod-                6 sekund
;       Oryginalny kod ASM      36 sekund
;       Borland Pascal v7.0     45 sekund
;       Borland C++ v4.02       29 sekund
;       Microsoft C++ v8.00     21 sekund
;
;       < Większość pomijanego kodu znajduje się tutaj, zobacz poprzednią wersję >

                print
                byte    „Wyniki obliczane”, cr,lf,0

; dla h := 1 to iterations

                mov     h, 1

hloop:

; Kopiowanie danych wejściowych do bufora wyjściowego
; Optymalizacja krok 31: Zastąpienie instrukcją movs

                push    ds.
                mov     ax, OutSeg
                mov     ds., ax
                mov     ax, InSeg
                mov     es, ax
                lea     si, DataOut
                lea     di, DataIn
                lea     di, DataIn
                mov     cx, (251*256) / 4
        rep     movsd
                pop     ds

; Optymalizacja krok #1: Konwersja pętli do postaci repeat…until
; for i := 1 to 249
                mov     I, 1
iloop:
; for j:=1 to 254
                mov     j, 1
jloop:

; Optymalizacja. Rozwinięcie dwóch pętli  najskrytszych:

                mov     bh, byte ptr i                  ;i jest zawsze mniejsze niż 256
                mov     bl, byte ptr j                  ; Obliczanie I*256+j!

                push    ds.
                mov     ax, InSeg                       ; korzyść z dostępu do InSeg
                mov     ds., ax

                mov     cx, 0                           ;tu obliczamy sumę
                mov     ah, ch
                mov     cl, ds.: DataIn[bx – 257]       ;DatIn[I-1][j-1]
                mov     al, ds: dataIn[bx – 256]        ;DataIn[I-1][j-1]
                add     cx, ax
                mov     al, ds: DataIn[bx-255]          ;DataIn[I-1][j+1]
                add     cx, ax
                mov     al, ds: DataIn[bx-1]            ;DataIn[I][j-1]
                add     cx, ax
```

```
                mov     al, ds: DataIn[bx+1]            ;DataIn[i][j+1]
                add     cx, ax
                mov     al, ds: DataIn[bx+255]          ;DataIn[I+1][j-1]
                add     cx, ax
                mov     al, ds:DataIn[bx+256]           ;DataIn[I+1][j]
                add     cx, ax
                mov     al, ds:DataIn[bx+257]           ; DataIn[i+1][j+1]
                add     cx, ax

                mov     al, ds:DataIn[bx]               ;DataIn[I][j]
                shl     ax, 3                           ;DataIn[I][j]*8
                add     cx, ax
                shr     cx, 4                           ;dzielenie przez 16
                mov     ax, OutSeg
                mov     ds, ax
                mov     ds:DataOut[bx], cl
                pop     ds

                inc     j
                cmp     j ,254
                jbe     jloop

                inc     i
                cmp     i, 249
                jbe     iloop

                inc     h
                mov     ax, h
                cmp     ax, Iterations
                jnbe    Done
                jmp     hloop

Done:           print
                Byte    “Zapisywanie wyniku”, cr, lf, 0

;               <Więcej opuszczonego kodu jest tutaj, zobacz poprzednią wersję>
```

Druga ,powyższa wersja używa jeszcze zmiennych pamięciowych dla większości obliczeń. Optymalizacja stosowana do kodu oryginalnego byłą głównie optymalizacją niezależną od języka. Kolejnym krokiem było zastosowanie jakiegoś określonego języka asemblera optymalizującego kod. Pierwsza optymalizacja musiała przesuwać wiele zmiennych do zbioru rejestrów 80x86. Poniższy kod dostarcza takiej optymalizacji. Chociaż poprawa jedynie czas wykonania o 2 sekundy, jest to poprawa o 33% (z sześciu do czterech sekund)!

```
; IMGPRCS.ASM
;
; Program przetwarzający obrazek
; Program ten rozmywa obrazek w ośmiobitowej skali szarości poprzez piksela w tym obrazku z ośmioma
; pikselami wokół. Średnia jest obliczana z (CurCell*8 + pozostałe 8 komórek)/ 16 , obciążając aktualną
;  komórkę o 50%.
;
; Ponieważ rozmiar obrazka to prawie 64 K, matryce wejściowa i wyjściowa są w różnych segmentach.
;
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
; Wersja #2: Trzy główne optymalizacje. (1) używa instrukcji movsd zamiast pętli do kopiowania danych z
; DataOut z powrotem do DataIn. (2) Używa formy repeat...until dla wszystkich pętli (3) rozwija najskrytsze
; dwie pętle (które są odpowiedzialne za większość poprawiania wydajności)
; Wersja #3: Użycie rejestrów dla wszystkich zmiennych. Ustawia rejestry segmentu raz dla wszystkich
; wykonań głównej pętli wiec kod nie musi ponownie ładować ds. za każdym razem. Oblicza indeks do każdego
```

```
; wiersza tylko raz (na zewnątrz pętli j)
;
;       Porównanie wydajności (system 66MHz 80486 DX/2)
;
;       Ten kod-                4 sekundy
;       1 optymalizacja         6 sekund
;       Oryginalny kod ASM      36 sekund
;
;       <Większość usuniętego kodu  stąd>

                print
                byte    „Obliczanie wyniku”,cr,lf,0

; Kopiowanie danej wejściowej do bufora wyjściowego

hloop:          mov     ax, InSeg
                mov     es, ax
                mov     ax, OutSeg
                mov     ds, ax
                lea     si, DataOut
                lea     di, DataIn
                mov     cx, (251*256) / 4
        rep     movsd

                assume  ds:InSeg, es:OutSeg
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax
                mov     cl, 249
iloop:          mov     bh, cl                  ;i*256
                mov     bl, 1                   ; start przy j =1
                mov     ch, 254                 ; # ilości pętli

jloop:
                mov     dx, 0                   ;tu obliczamy sumę
                mov     ah, dh
                mov     dl, DataIn[bx-257]      ;DataIn[i-1][j-1]
                mov     al, DataIn[bx-256]      ;DataIn[I-1][j]
                add     dx, ax
                mov     al, dataIn[bx-255]      ;DatIn[i-1][j+1]
                add     dx, ax
                mov     al, DataIn[bx-1]        ;DataIn[I][j-1]
                add     dx, ax
                mov     al, DataIn[bx+1]        ;DataIn[i][j+1]
                add     dx, ax
                mov     al, DataIn[bx+255]      ;DataIn[i+1][j-1]
                add     dx, ax
                mov     al, DataIn[bx+256]      ;DataIn[I+j][j]
                add     dx, ax
                mov     al, DataIn[bx+257]      ;DataIn[I+1][j+1]
                add     dx, ax

                mov     al, DataIn[bx]          ;DataIn[i[j]
                shl     ax, 3                   ;DataIn[I][j]*8
                add     dx ,ax
                shr     dx, 4                   ;dzielenie przez 16
                mov     DataOut[bx], dl
```

```
                inc     bx
                dec     ch
                jne     jloop

                dec     cl
                jne     iloop

                dec     bp
                jne     hloop
Done:           print
                byte    "Zapisywanie wyniku", cr, lf, 0

; < Więcej usuniętego kodu znajduje się tutaj >
```

Zauważmy ,że przy każdej iteracji. Powyższy kod kopiuje daną wyjściową z powrotem jako daną wejściową. Jest to prawie 6 i pól megabajta danych przesuniętych dla 100 iteracji!!! Poniższa wersja programu rozmywającego rozwija dwukrotnie hloop. Pierwsze wystąpienie kopiuje dane z DataIn do DataOut podczas obliczania rozmycia, druga instancja kopiuje dane z DataOut z powrotem do DataIn podczas rozmywania obrazu. Stosując te dwie sekwencje kodu, program zachowuje kopiowanie danych z jednego punktu do innego. Wersja ta również utrzymuje popularne obliczanie pomiędzy dwoma sąsiadującymi komórkami zachowując kilka instrukcji w pętli najskrytszej. Wersja ta układa instrukcje w pętli najskrytszej pomagając uniknąć hazardu danych na procesorze 80486 i późniejszych. Końcowy wynik jest prawie 40% szybszy niż wersji poprzedniej (w dół o 2,5 sekundy z czterech sekund).

```
; IMGPRCS.ASM
;
; Program przetwarzający obrazek
; Program ten rozmywa obrazek w ośmiobitowej skali szarości poprzez piksela w tym obrazku z ośmioma
; pikselami wokół. Średnia jest obliczana z (CurCell*8 + pozostałe 8 komórek)/ 16 , obciążając aktualną
;  komórkę o 50%.
;
; Ponieważ rozmiar obrazka to prawie 64 K, matryce wejściowa i wyjściowa są w różnych segmentach.
;
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
; Wersja #2: Trzy główne optymalizacje. (1) używa instrukcji movsd zamiast pętli do kopiowania danych z
; DataOut z powrotem do DataIn. (2) Używa formy repeat...until dla wszystkich pętli (3) rozwija najskrytsze
; dwie pętle (które są odpowiedzialne za większość poprawiania wydajności)
; Wersja #3: Użycie rejestrów dla wszystkich zmiennych. Ustawia rejestry segmentu raz dla wszystkich
; wykonań głównej pętli wiec kod nie musi ponownie ładować ds. za każdym razem. Oblicza indeks do każdego
; wiersza tylko raz (na zewnątrz pętli j)
; Wersja #4: Eliminuje kopiowanie danych z DataOut do DataIn w każdym kroku. Usuwa hazardy. Utrzymuje
; pdwyrażenia
;       Porównanie wydajności (system 66MHz 80486 DX/2)
;
;       Ten kod-                2,5undy
;       2optymalizacja          4 sekund
;       1 optymalizacja         6 sekund
;       Oryginalny kod ASM      36 sekund


;       <Więcej usuniętego kodu tu, zobacz wersję oryginalną >

                print
                byte    „Obliczanie wyniku", cr, lf,0

                assume  ds.:InSeg, es:OutSeg

                mov     ax, InSeg
                mov     ds., ax
```

```
                mov     ax, OutSeg
                mov     es, ax

; Kopiowanie danych raz, więc uzyskujemy brzegi w obu tablicach

                mov     cx, (251*256) / 4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd
; „hloop” powtarzana raz dla każdej iteracji

hloop:
                mov     ax, InSeg
                mov     ds., ax
                mov     ax, OutSeg
                mov     es, ax

; „iloop” przetwarza wiersze w matrycach
                mov     cl, 249
iloop:          mov     bh, cl                  ;i *256
                mov     bl, 1                   ; zaczynamy od j =1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0                   ;tu obliczamy sumę
                mov     bh, 0
                mov     ah, 0
; „jloop” przetwarza pojedyncze elementy tablicy. Pętla ta będzie rozwinięta raz  pozwalając podzielić
; na dwie części obliczenia

jloop:

; Suma DataIn[i-1[j] + DataIn[i-1][j+1]+DataIn[i+1][j] + DataIn[i+1][j+1] będzie używana w drugiej
; połowie obliczeń. Wiec zachowujemy jej wartość w rejestrze (di) dopóki nie będziemy jej potrzebować

                mov     dl, DataIn[si-256]      ;[i-1,j]
                mov     al, DataIn[si-255]      ;[i-1, j+1]
                mov     bl, DataIn[si+257]      ;[I+1, j+1]
                add     dx, ax
                mov     al, DataIn[si+256]      ;[i+1,j]
                add     dx, bx
                mov     bl, DataIn[si+1]        ;[i,j+1]
                add     dx, ax
                mov     al, DataIn[si+255]      ;[i+1, j-1]

                mov     di, dx                  ;Zachowanie częściowych wyników

                add     dx, bx
                mov     bl, DataIn[si-1]        ;[i,j-1]
                add     dx ,ax
                mov     al, dataIn[si]          ;[i,j]
                add     dx, bx
                mov     bl, DataIn[si-257]      ;[I-1,j-1]
                shl     ax, 3                   ;DataIn[I,j*8
                add     dx, bx
                add     dx, ax
                shr     ax, 3                   ;przywrócenie DataIn[i,j]
                shr     dx, 4                   ;dzielenie przez 16
                add     di, ax
                mov     DataOut[si], di
```

```
; Okay, przetwarzamy kolejną komórkę. Zauważmy, że mamy już  częściową sumę usytuowaną  w DI.
; Nie zapomnijmy, że nie mamy SI w tym miejscu. (To jest druga połówka rozwijanej pętli)

                mov     dx, di                          ;suma częściowa
                mov     bl, DataIn[si-254]              ;[i-1, j+1]
                mov     al, DataIn[si+2]                ;[i, j+1]
                add     dx, bx
                mov     bl, DataIn[si+258]              ;[i+1,j+1]
                add     dx, ax
                mov     al, DataIn[si+1]                ;[I,j]
                add     dx, bx
                shl     ax, 3                           ;DataIn[I][j]* 8
                add     si, 2
                add     dx ,ax
                mov     ah, 0                           ;zerowanie dla kolejnej iteracji
                shr     dx, 4                           ;dzielenie przez 16
                dec     ch
                mov     DataOut[si-1], dl
                jne     jloop
                dec     cl
                jne     iloop

                dec     bp
                je      Done

; Specjalny przypadek, więc nie musimy przesuwać danej pomiędzy dwoma tablicami. Jest  to wersja pętli
; rozwijanej hloop, która zamienia wejście i wyjście tablic , więc nie musimy przesuwać danej w pamięci

                mov     ax, OutSeg
                mov     ds., ax
                mov     ax, InSeg
                mov     es, ax
                assume  es:InSeg, ds.: OutSeg

hloop2:
                mov     cl, 249
iloop2:         mov     bh, cl
                mov     bl, 1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0
                mov     bh, 0
                mov     ah, 0
jloop2:
                mov     dl, DataOut[si-256]
                mov     al, dataOut[si-255]
                mov     bl, DataOut[si+257]
                add     dx ,ax
                mov     al, DataOut[si+256]
                add     dx, bx
                mov     bl, DataOut[si+1]
                add     dx, ax
                mov     al, DataOut[si+255]

                mov     di, dx

                add     dx, bx
                mov     bl, DataOut[si-1]
                add     dx, ax
```

```
                mov     al, DataOut[si]
                add     dx, bx
                mov     bl, DatOut[si-257]
                shl     ax, 3
                add     dx, bx
                add     dx ,ax
                shr     ax, 3
                shr     dx, 4
                mov     DataIn[si], dl
                mov     dx, di
                mov     bl, DataOut[si-254]
                add     dx, ax
                mov     al, DataOut[si+2]
                add     dx, bx
                mov     bl, DataOut[si+258]
                add     dx, ax
                mov     al,  dataOut[si+1]
                add     dx, bx
                shl     ax, 3
                add     si, 2
                add     dx, ax
                mov     ah, 0
                shr     dx, 4
                dec     ch
                mov     DataIn[si-1], dl
                jne     jloop2
                dec     cl
                jne     iloop2
                dec     bp
                je      Done2
                jmp     hloop

; Łata gwarantująca, że dana zawsze znajduje się w segmencie wyjściowym

Done2:
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax
                mov     cx, (251*256) / 4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd

Done:           print
                Byte    "Zapisanie wyników", cr, lf,0

;       <pozostały usunięty kod mamy tu, zobacz program oryginalny>
```

Kod ten dostarcza dobrego przykładu tego rodzaju optymalizacji, którego boi się większość ludzi .jest wiele cykli obliczeń, szeregowanie instrukcji i innych szalonych rzeczy, które czynią program trudniejszym do odczytu i zrozumienia. Jest to rodzaj optymalizacji, z którego są znani programiści asemblerowi; rzecz , która daje początek zdaniu „nigdy nie optymalizuj wcześnie" . Nie powinniśmy nigdy próbować tego typu optymalizacji dopóki nie wyczerpiemy wszystkich innych możliwości. Pisząc nasz kod w taki sposób, uczynimy go bardzo trudnym dla dokonywania dalszych zmian w nim. Nawiasem mówiąc, powyższy kod zabiera około 15 godzin dla rozwinięcia i zdebuggowania (debuggowanie zabiera najwięcej czasu) Przyczyni się to do poprawy o 0.1 sekundy (dla 100 iteracji) na każdą godzinę pracy. Chociaż kod ten z pewnością nie jest w pełni optymalny, trudno jest uzasadnić próby zajmowania więcej czasu na poprawianie tego kodu przez mechaniczne działania (np. przesuwanie instrukcji itp.) ponieważ wydajność może poprawi się odrobinę.

W powyższych czterech krokach zredukowaliśmy czas działania kodu asemblerowego z 36 sekund do 2.5 sekundy. Całkiem imponujący wyczyn. Jednakże, nie powinniśmy uwierzyć ,że był to łatwy sposób lub nawet ,że były to tylko cztery zawiłe kroki Podczas faktycznego rozwoju tego przykładu, było wiele prób, które nie poprawiły wydajności (faktycznie, niektóre modyfikacje redukowały wydajność) a inne nie poprawiały wydajności wystarczająco uzasadniając ich wdrożenie. Aby zademonstrować ten ostatni punkt, poniższy kod zawiera główne zmiany w sposobie organizacji danych. Pętla główna działa na obiektach 16 bitowych w pamięci zamiast obiektach 8 bitowych,. W niektórych maszynach z dużym zewnętrznym cachem (256K lub lepszym) algorytm ten dostarcza drobnej poprawy wydajności (2.4 sekundy z 2.5 sekundy). Jednakże, na innej maszynie działa wolniej. Dlatego też, kod ten nie może być traktowany jako końcowa implementacja:

```
; IMGPRCS.ASM
;
; Program przetwarzający obrazek
; Program ten rozmywa obrazek w ośmiobitowej skali szarości poprzez piksela w tym obrazku z ośmioma
; pikselami wokół. Średnia jest obliczana z (CurCell*8 + pozostałe 8 komórek)/ 16 , obciążając aktualną
;  komórkę o 50%.
;
; Ponieważ rozmiar obrazka to prawie 64 K, matryce wejściowa i wyjściowa są w różnych segmentach.
;
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
; Wersja #2: Trzy główne optymalizacje. (1) używa instrukcji movsd zamiast pętli do kopiowania danych z
; DataOut z powrotem do DataIn. (2) Używa formy repeat...until dla wszystkich pętli (3) rozwija najskrytsze
; dwie pętle (które są odpowiedzialne za większość poprawiania wydajności)
; Wersja #3: Użycie rejestrów dla wszystkich zmiennych. Ustawia rejestry segmentu raz dla wszystkich
; wykonań głównej pętli wiec kod nie musi ponownie ładować ds. za każdym razem. Oblicza indeks do każdego
; wiersza tylko raz (na zewnątrz pętli j)
; Wersja #4: Eliminuje kopiowanie danych z DataOut do DataIn w każdym kroku. Usuwa hazardy. Utrzymuje
; podwyrażenia
; Wersja #5; Konwertuje tablicę danych do słów zamiast bajtów i działają na wartościach 16 bitowych. Dają
; minimalne przyspieszenie
;
;       Porównanie wydajności (system 66 MHz 80486 DX/2)
;
;       Ten kod                 2,4 sekundy
;       3 optymalizacja         2.5 sekundy
;       2 optymalizacja         4 sekundy
;       1 optymalizacja         6 sekund
;       Oryginalny kod ASM      36 sekund

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list
                .386
                option          segment :use16
dseg            segment para public 'data'

ImgData         byte    251  dup (256 dup (?))
InName          byte    „roller1.raw", 0
OutName         byte    „roller2.raw", 0
Iterations      word    0

dseg            ends

; Kod ten czyni niestosowne założenie, że poniższe segmenty są ładowane sąsiednio w pamięci! Również,
; ponieważ te segmenty są wyrównane paragrafem, kod ten zakłada , że segmenty te będą zawierały pełne
; 65, 536 bajtów. Nie możemy zadeklarować segmentu z dokładnie 65,536 bajtami w MASM. Jednakże,
; opcja wyrównania paragrafem zakłada ,że te ekstra bajty są dodawane na końcu każdego segmentu.
```

```
DataSeg1        segment para  public ‘ds1’
Data1a          byte      65535  dup (?)
DataSeg         ends

DataSeg         segment para public ‘ds2’
Data1b          byte      65536  dup (?)
DataSeg2        ends

DataSeg3        segment  para  piblic ‘ds3’
Data2a          byte      65535 dup (?)
DataSeg3        ends

DataSeg4        segment para  public ‘ds4’
Data2b          byte      65535  dup  (?)
DataSeg4        ends

cseg            segment para  public ‘code’
                assume  cs: cseg, ds: dseg

Main            proc
                mov       ax, dseg
                mov       ds, ax
                meminit

                mov       ax, 3d00h                     ;otwarcie pliku do odczytu
                lea       dx, InName
                int       21h
                jnc       GoodOpen
                print
                byte      „Nie można otworzyć pliku wejściowego”, cr, lf, 0
                jmp       Quit

GoodOpen:       mov       bx, ax                        ;uchwyt pliku
                lea       dx, ImgData
                mov       cx, 256 * 251                 ; rozmiar pliku danych do odczytu
                mov       ah, 3Fh
                int       21h
                cmp       ax, 256*251                   ;zobacz czy możemy czytać dane
                je        GoodRead
                print
                byte      „Nie odczytano pliku poprawnie”, cr, lf, 0
                jmp       Quit

GoodRead:       print
                byte      „Wprowadź liczbę iteracji: „,0
                getsm
                atoi
                free
                mov       Iterations, ax
                cmp       ax, 0
                jle       Quit

                printf
                byte      „Wynik obliczony dla %d iteracji”, cr, lf, 0
                dword     Iterations

; Kopiujemy daną i rozszerzamy ją z ośmiu do szesnastu bitów. Pierwsza pętla obsługuje 32,785 bajtów,
; druga pętla obsługuje pozostałe  bajty
```

```
                mov     ax, DataSeg1
                mov     es, ax
                mov     ax, DatSeg3
                mov     fs, ax

                mov     ah, 0
                mov     cx, 32768
                lea     si, ImgData
                xor     di ,di                  ;dana wyjściowa jest pod offsetem zero
CopyLoop:       lodsb                           ;odczyt bajtu
                mov     fs:[di], ax             ;przechowanie słowa w DataSeg3
                stosw                           ;przechowanie słowa w DataSeg1
                dec     cx
                jne     CopyLoop

                mov     di, DataSeg2
                mov     es, di
                mov     di, DataSeg4
                mov     fs, di
                mov     cx, (251* 256) – 32768
                xor     di ,di
CopyLoop1:      lodsb                           ;odczyt bajtu
                mov     fs:[di], ax             ;przechowanie słowa w DataSeg4
                stosw                           ;przechowanie słowa w DataSeg2
                dec     cx
                jne     CopyLoop1

; hloop kończy jedną iterację przesunięciem danych z Data1a/Data1b do Data2a/Data2b

hloop:          mov     ax, DataSeg1
                mov     ds, ax
                mov     ax, DataSeg3
                mov     es, ax

; przetwarzanie pierwszych 127 wierszy (65,024 bajtów) tablicy:

                mov     cl ,127
                lea     si, Data1a+202h         ;start pod [1,1]
iloop0:         mov     ch, 254/2
jloop0:         mov     dx, [si]                ;[i,j]
                mov     bx, [si – 200h]         ;[i-1, j]
                mov     ax, dx
                shl     dx, 3                   ;[i,j]* 8
                add     bx, [si-1feh]           ;[i-1, j+1]
                mov     bp, [si+2]              ;[i,j+1]
                add     bx, [si+200h]           ;[I+1,j]
                add     dx, bp
                add     bx, [si+202h]           ;[i+1, j+1]
                add     dx, [si-202h]           ;[i-1, j-1]
                mov     di, [si-1fch]           ;[i-1, j+2]
                add     dx, [si-2]              ;[I, j-1]
                add     di, [si+4]              ;[i, j+2]
                add     dx, [si+1feh]           ;[i+1, j-1]
                add     di, [si+204h]           ;[i+1, j+2]
                shl     bp, 3                   ;[i, j+1]*8
                add     dx, bx
                add     bp, ax
                shr     dx, 4                   ;dzielenie przez 16
                add     bp, bx
```

```
                mov     es:[si], dx             ;przechowanie wejścia [i, j]
                add     bp, di
                add     si, 4                   ;wpływ kolejnej operacji przechowania
                shr     bp, 4                   ;dzielenie przez 16
                dec     ch
                mov     es:[si-2], bp           ;przechowanie wejścia [i,j+1]
                jne     jloop0

                add     si, 4                   ;przeskok do początku kolejnego wiersza
                dec     cl
                jne     iloop0

; Przetwarzanie ostatnich 124 wierszy tablicy. Wymaga to tego ,ze przełączamy z jednego segmentu do
; kolejnego .Zauważmy ,że segmenty nakładają się

                mov     ax, DataSeg2
                sub     ax, 40h                 ;robienie kopii ostatnich dwóch wierszy w DS2
                mov     ds., ax
                mov     ax, DataSeg4
                sub     ax, 40h                 ;robienie kopii ostatnich dwóch wierszy w DS4
                mov     es, ax

                mov     cl, 251 – 127-1         ;pozostałe wiersze do przetworzenia
                mov     si, 202h                ; kontynuacja z kolejnymi wierszami
iloop1:         mov     ch, 254/2
jloop1:         mov     dx, [si]                ;[i,j]
                mov     bx, [si-200h]           ;[i-1,j]
                mov     ax, dx
                shl     dx, 3                   ;[i,j]*8

                add     bx, [si-1feh]           ;[i-1,j+1
                mov     bp, [si+2]              ;[I,j+1]
                add     bx, [si+200h]           ;[I+1,j]
                add     dx, bp
                add     bx, [si+202h]           ;[i+1, j+1]
                add     dx, [si-202h]           ;[i-1,j-1]
                mov     di, [si-1fch]           ;[i-1, j+2]
                add     dx, [si-2]              ;[i, j-1]
                add     di, [si+4]              ;[i,j+2]
                add     dx, [si+1feh]           ;[i+1,j-1]
                add     di, [si+204h]           ;[i+1, j+2]
                shl     bp,3                    ;[i,j+1]*8
                add     dx, bx
                add     bp, ax
                shr     dx, 4                   ;dzielenie przez 16
                add     bp, bx
                mov     es:[si], dx             ; przechowanie wejścia [i,j]
                add     bp, di
                add     si ,4                   ;wpływ na kolejną operację przechowania
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp           ; przechowanie wejścia [i, j+1]
                jne     jloop1

                add     si, 4                   ;skok do początku kolejnego wiersza
                dec     cl
                jne     iloop1

                mov     ax, dseg
```

```
                mov     ds., ax
                assume  ds.:dseg
                dec     Iterations
                je      Done0

; Rozwijamy  pętle iteracji więc możemy przesunąć dane z DataSeg 2/4 z powrotem do datSeg1/3 bez
; marnowania dodatkowego czasu. Inny niż kierunek przesuwania danych, kod ten jest  praktycznie identyczny
; do powyższego

                mov     ax, DataSeg3
                mov     ds., ax
                mov     ax, DataSeg1
                mov     es, ax

                mov     cl, 127
                lea     si, Data1a+202h
iloop2:         mov     ch, 254/2
jloop2:         mov     dx, [si]
                mov     bx, [si-200h]
                mov     ax, dx
                shl     dx, 3
                add     bx,  [si-1feh]
                mov     bp, [si+2]
                add     bx, [si+200h]
                add     dx, bp
                add     bx, [si+202h]
                add     dx, [si-202h]
                mov     di, [si-1fch]
                add     dx, [si-2]
                add     di, [si+4]
                add     dx, [si+1feh]
                add     di, [si+204h]
                shl     bp, 3
                add     dx, bx
                add     bp, ax
                shr     dx, 4
                add     bp, bx
                mov     es:[si], dx
                add     bp, di
                add     si, 4
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp

                jne     jloop2
                add     si, 4
                dec     cl
                jne     iloop2
                mov     ax, DataSeg4
                sub     ax, 40h
                mov     ds, ax
                mov     ax, DataSeg2
                sub     ax, 40h
                mov     es, ax
                mov     cl, 251-127-1
                mov     si, 202h
iloop3:         mov     ch, 254/2
jloop3:         mov     dx, [si]
                mov     bx. [si-200h]
```

```
                mov     ax, dx
                shl     dx, 3
                add     bx, [si-1feh]
                mov     bp, [si+2]
                add     bx, [si+200h]
                add     dx, bp
                add     bx, [si+202h]
                add     dx, [si-202h]
                mov     di, [si-1fch]
                add     dx, [si-2]
                add     di, [si+4]
                add     dx, [si+1feh]
                add     di, [si+204h]
                shl     bp, 3
                add     dx, bx
                add     bp, ax
                shr     dx, 4
                add     bp, bx
                mov     es:[si], dx
                add     bp, di
                add     si, 4
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp
                jne     jloop3
                add     si, 4
                dec     cl
                jne     iloop3
                mov     ax, dseg
                mov     ds, ax
                assume  ds:dseg
                dec     Iterations
                je      Done2
                jmp     hloop
Done2:          mov     ax, DataSeg1
                mov     bx, DataSeg2
                jmp     Finish
Done0:          mov     ax, DataSeg3
                mov     bx, DataSeg4
Finish:         mov     ds, ax
                print
                byte    “zapisywanie wyniku”, cr,lf, 0

; Konwersja danych powrotnie do bajtu i za[psia do pliku wyjściowego:

                mov     ax, dseg
                mov     es, ax
                mov     cx, 32768
                lea     di, ImgData
                xor     si, si                  ;Dana wyjściowa jest pod offsetem zero
CopyLoop3:      lodsw                           ;odczyt słowa z tablicy końcowej
                stosb                           ;zapis bajtu do tablicy wyjściowej
                dec     cx
                jne     CopyLoop3

                mov     ds., bx
                mov     cx, (251*256) – 32768
                xor     si, si
CopyLoop4:      lodsw                           ;odczyt końcowej danej słowa
```

```
                stosb                                   ;zapis bajtu  danej do tablicy wyjściowej
                dec     cx
                jne     CopyLoop4

; Okay, zapis danej do pliku wyjściowego

                mov     ah, 3ch                         ;tworzenie pliku wyjściowego
                mov     cx, 0                           ;normalny atrybut pliku
                mov     dx, dseg
                mov     ds., dx
                lea     dx, OutName
                int     21h
                jnc     GoodCreate
                print
                byte    „Nie można stworzyć pliku wyjściowego”, cr, lf,0
                jmp     Quit

GoodCreate:     mov     bx ,ax                          ;uchwyt pliku
                push    bx
                mov     dx, dseg                        ;gdzie dana może zostać znaleziona
                mov     ds, dx
                lea     dx, ImgData
                mov     cx, 256*251                     ;rozmiar danej do zapisu
                mov     ah, 40h                         ;operacja zapisu
                int     21h
                pop     bx                              ;przywrócenie uchwyty do zamknięcia
                cmp     ax, 256*251                     ;zobacz czy zapisano dane
                je      GoodWrite
                print
                byte    „Nie zapisano poprawnie pliku”, cr, lf,0
                jmp     Quit

GoodWrite:      mov     ah, 3eh                                 ;operacja zamknięcia
                int     21h
Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment para stack ‘stack’
stk             byte    1024  dup (“stack”)
sseg            ends
zzzzzzseg       segment para public ‚zzzzzz’
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

Oczywiście, absolutnie najlepszym sposobem poprawy wydajności każdego kawałka kodu jest lepszy algorytm. Wszystkie powyższe wersje asemblerowe były ograniczone przez pojedyncze wymaganie – musiały tworzyć taki sam plik jak oryginalny program pascalowski. Powyższy przykład optymalizacji jest doskonałym przykładem .Kod asemblerowy wiernie zachowuje semantykę oryginalnego programu Pascala; oblicza średnią ważoną wszystkich wewnętrznych pikseli jako sumę ośmiu sąsiadujących pikseli plus osiem razy bieżącą wartość piksela z całą sumą dzieloną przez16. Teraz jest to dobra funkcja rozmywająca, ale nie jest to jedynie funkcja rozmywająca .Użytkownik Photoshop (lub innego programu przetwarzania obrazu) nie musi martwić się o algorytm jako taki. Kiedy użytkownik wybiera „rozmycie obrazu” chce go jako wyjście z ostrości. Dokładnie jak dużo ostrości jest generalnie nie istotne. Faktycznie, mniejsza jest lepsza ponieważ użytkownik może zawsze uruchomić algorytm rozmycia ponownie (lub określić jakąś liczbę iteracji) poniższy program asemblerowy pokazuje jak uzyskać lepszą wydajność poprzez zmodyfikowanie algorytmu rozmycia redukując liczbę instrukcji potrzebnych do wykonania najskrytszych pętli Oblicza rozmycie poprzez uśrednienie piksela z

czterema sąsiednimi, powyżej poniżej, na lewo i na prawo od bieżącego piksela. Ta modyfikacja dostarcza programu, który uruchamia 100 iteracji w 2,2 sekundy z 12% poprawą w stosunku do poprzednich wersji:

```
; IMGPRCS.ASM
;
; Program przetwarzający obrazek
; Program ten rozmywa obrazek w ośmiobitowej skali szarości poprzez piksela w tym obrazku z ośmioma
; pikselami wokół. Średnia jest obliczana z (CurCell*8 + pozostałe 8 komórek)/ 16 , obciążając aktualną
;  komórkę o 50%.
;
; Ponieważ rozmiar obrazka to prawie 64 K, matryce wejściowa i wyjściowa są w różnych segmentach.
;
; Wersja #1: Proste tłumaczenie z Pascala na Asembler
; Wersja #2: Trzy główne optymalizacje. (1) używa instrukcji movsd zamiast pętli do kopiowania danych z
; DataOut z powrotem do DataIn. (2) Używa formy repeat...until dla wszystkich pętli (3) rozwija najskrytsze
; dwie pętle (które są odpowiedzialne za większość poprawiania wydajności)
; Wersja #3: Użycie rejestrów dla wszystkich zmiennych. Ustawia rejestry segmentu raz dla wszystkich
; wykonań głównej pętli wiec kod nie musi ponownie ładować ds. za każdym razem. Oblicza indeks do każdego
; wiersza tylko raz (na zewnątrz pętli j)
; Wersja #4: Eliminuje kopiowanie danych z DataOut do DataIn w każdym kroku. Usuwa hazardy. Utrzymuje
; podwyrażenia
; Wersja #6: Zmiana algorytmu rozmycia używając kilku obliczeń. Wersja ta  NIE tworzy takich samych danych
;jak inne programy
;
;       Porównanie wydajności (system 66 MHz 80486 DX/2)
;
;       Ten kod                 2,2sekundy
;       3 optymalizacja         2.5 sekundy
;       2 optymalizacja         4 sekundy
;       1 optymalizacja         6 sekund
;       Oryginalny kod ASM      36 sekund

;       <Reszta usuniętego kodu tutaj, zobacz oryginalny program>

                print
                byte    „Obliczenie wyniku”, cr,lf,0

                assume  ds.: InSeg, es:OutSeg

                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax

;Kopiowanie danej raz więc uzyskujemy brzegi w obu tablicach

                mov     cx, (251*256)/4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd

; “hloop” powtarzana raz dla każdej iteracji

hloop:
                mov     ax, InSeg
                mov     ds., ax
                mov     ax, OutSeg
                mov     es, ax
```

; „iloop” przetwarza wiersze w matrycach

```
                mov     cl, 249
illop:          mov     bh, cl                  ;i*256
                mov     bl, 1                   ;star przy j = 1
                mov     ch, 254 /2
                mov     si, bx
                mov     dh, 0                   ;tu obliczam sumę
                mov     bh, 0
                mov     ah, 0
```

; „jloop” przetwarza pojedyncze elementy tablicy. Pętla ta rozwijana jest raz pozwalając podzielić obleczenie na
; dwie części.

jloop:

; Suma DataIn[i-1][j]+ DataIn[i-1][j+1]+DataIn[i+1][j]+DataIn[i+1][j+1] będzie użyta w drugiej połówce tego
; obliczenia. Więc zapisujemy tą wartość w rejestrze (di) póki jej nie będziemy potrzebowali

```
                mov     dl, dataIn[si]          ;[I,j]
                mov     al, DataIn[si-256]      ;[i-1, j]
                shl     dx, 2                   ;[i,j]*4
                mov     bl, DataIn[si-1]        ;[I,j-1]
                add     dx ,ax
                mov     al, dataIn[si+1]        ; [I,j+1]
                add     dx, bx
                mov     bl, DataIn[si+256]      ;[I+1,j]
                add     dx, ax
                shl     ax,2                    ;[I,j+1]*4
                add     dx, bx
                mov     bl, dataIn[si-255]              ;[I-1,j+1]
                shr     dx, 3                           ;dzielenie przez 8
                add     ax, bx
                mov     DataOut[si], dl
                mov     bl, dataIn[si+2]                ;[i,j+2]
                mov     dl, DataIn[si+257]              ;[I+1,j+1]
                add     ax, bx
                mov     bl, dataIn[si]                  ;[i,j]
                add     ax, dx
                add     ax, bx
                shr     ax, 3
                dec     ch
                mov     DataOut[si+1], al
                jne     jloop
                dec     cl
                jne     iloop
                dec     bp
                je      Done
```

; Specjalny przypadek, wiec nie musimy przesuwać danej pomiędzy dwoma tablicami. Jest to rozwijana wersja hloop, która zmienia tablicę wejściową i wyjściową więc nie musimy danej w pamięci

```
                mov     ax, OutSeg
                mov     ds., ax
                mov     ax, InSeg
                mov     es, ax
                assume  es:InSeg, ds.:OutSeg
```

hloop2:

```
                mov     cl, 249
iloop2:         mov     bh, cl
                mov     bl, 1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0
                mov     bh, 0
                mov     ah, 0

jloop2:
                mov     dl, dataOut[si-256[
                mov     al, DataOut[si-255]
                mov     bl, DataOut[si+257]
                add     dx, ax
                mov     al, DataOut[si+256]
                add     dx, bx
                mov     bl, DatOut[si+1]
                add     dx, ax
                mov     al, DataOut[si+255]

                mov     di, dx
                add     dx, bx
                mov     bl, DataOut[si-1]
                add     dx, ax
                mov     al, DataOut[si]
                add     dx, bx
                mov     bl, dataOut[si-257]
                shl     ax, 3
                add     dx, bx
                add     dx, ax
                shr     ax, 3
                shr     dx, 4
                mov     DaatIn[si], dl
                mov     dx, di
                mov     bl, DataOut[si-254]
                add     dx, ax
                mov     al, dataOut[si+2]
                add     dx, bx
                mov     bl, datOut[si+258]
                add     dx, ax
                mov     al, DataOut[si+1]
                add     dx, bx
                shl     ax, 3
                add     si, 2
                add     dx, ax
                mov     ah, 0
                shr     dx, 4
                dec     ch
                mov     DataIn[si-1], dl
                jne     jloop2
                dec     cl
                jne     iloop2
                dec     bp
                je      Done2
                jmp     hloop

; Łatka gwarantuje ,że dana zawsze pozostaje w segmencie danych

Done2:
```

```
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax
                mov     cx, (251*256) /4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd
Done:           print
                byte    "Zapisanie wyniku", cr, lf, 0

;       < Pozostały usunięty kod tutaj, zobacz program oryginalny >
```

Jedną ważną rzeczą do zapamiętania o kodzie w tej sekcji jest to ,że optymalizujemy go dla 100 iteracji. Jednak okazuje się ,że te optymalizacje stosuje się równie dobrze do większej ilości iteracji. Co niekoniecznie może być prawdą dla kilu iteracji. W szczególności, jeśli uruchamiamy tylko jedną iterację, każde skopiowanie danej na koniec operacji ułatwi skonsumowanie dużej części czasu zachowanego przez optymalizację.
Ponieważ rzadko użytkownik będzie rozmywał obrazek 100 razy , nasza optymalizacja może nie być tak dobra jak można ją uczynić. Jednakże, sekcja ta dostarcza dobrego przykładu kroków jakie musimy wykonać aby zoptymalizować dany program. Sto iteracji było dobrym wyborem dla tego przykładu ponieważ było łatwiej zmierzyć czas działania wszystkich wersji programu. Jednak, musimy pamiętać ,że powinniśmy optymalizować nasze programy w przypadkach szczególnych a nie w każdym

---

25.6 PODSUMOWANIE

Program komputerowy działa często znacznie wolniej niż wymaga tego zadanie. Proces zwiększania szybkości programu jest znany jako optymalizacja Niestety optymalizacja jest trudnym i czasochłonnym zadaniem, czasami nie da się wykonać lekko. Wielu programistów często optymalizuje swoje programy zanim określa czy muszą to robić czy nie lub (co gorsza) optymalizują tylko część programu uznając ,ze muszą przepisać ten kod po zoptymalizowaniu. Inni, ignoranci, często zabierają się do optymalizacji złej sekcji programu .Ponieważ optymalizacja jest powolnym i trudnym procesem, chcemy spróbować i uczynić ,że optymalizujemy nasz kod tylko raz. Sugeruje to ,że optymalizacja powinna być ostatnim zadaniem, kiedy piszemy program.

Jedna szkoła mówi o filozofii grupy Późnej Optymalizacji. Ich argumentacją jest to, że zoptymalizowany program często niszczy czytelność i pielęgnowalność programu. Dlatego też powinno się tylko wybrać ten krok kiedy jest to absolutnie konieczne i tylko na końcowym etapie projektowanie programu.

Grupa Wczesnej Optymalizacji, z doświadczenia wie, że programy, które nie są napisane jako szybkie, często muszą być przepisane kompletnie aby uczynić je szybszymi. Dlatego też, oni często przyjmują postawę, że optymalizacja powinna mieć miejsce razem ze zwykłym projektowaniem programu. Generalnie, punkt widzenia grupy wczesnej optymalizacji jest daleko różniący się od grupy późnej optymalizacji. Grupa wczesnej optymalizacji zakłada ,że dodatkowy czas spędzony nad optymalizacją programu podczas projektowania wymaga mniej czasu niż zaprojektowanie programu a potem jego optymalizacja.

*:kiedy optymalizować, kiedy nie optymalizować"

Po napisaniu programu i określeniu, że działa zbyt wolno, kolejnym krokiem jest zlokalizowanie kodu który działa zbyt wolno. Po zidentyfikowaniu wolnej sekcji naszego programu, możemy popracować nad przyspieszeniem naszego programu. Zlokalizowanie 10% kodu, który zabiera 90% czasu wykonania nie zawsze jest łatwym zadaniem. Czterema powszechnymi technikami używanymi przez ludzi są prób i błędów, optymalizacji wszystkiego, analiza programu i analiza eksperymentalna (tj. użycie profilu). Znalezienie „gorących miejsc" pierwszym krokiem optymalizacji.

*"Jak znaleźć powolny kod w naszych programach"

Argumentem używanym przez ludzi przekonującym do późniejszej optymalizacji jest to ,że maszyny są szybkie więc optymalizacja jest rzadko konieczna. Chociaż ten argument jest często wyolbrzymiany, często jest prawdą, że wiele nie zoptymalizowanych programów działających dość szybko i nie wymagają żadnej optymalizacji dla zadawalającej wydajności. Z drugiej strony, programy, które działają lepiej mogą być zbyt wolne kiedy są uruchamiane z innym oprogramowaniem.

*”Czy optymalizacja jest konieczna?”

Są trzy formy optymalizacji jakie możemy zastosować dla poprawy wydajności programu: wybranie lepszego algorytmu, wybór lepszej implementacji algorytmu, lub „zliczanie cykli”. Wielu ludzi (zwłaszcza z grupy późnej optymalizacji) tylko rozważa przypadek późniejszej „optymalizacji”. To wstyd, ponieważ ostatni przypadek często tworzy najmniejszy przyrost poprawy wydajności.

*”Trzy typy optymalizacji”

Optymalizacja nie jest czymś czego możemy się nauczyć z książki. Potrzeba dużo doświadczenia i praktyki. Niestety, ci z niewielkim doświadczeniem praktycznym odkrywają, że ich wysiłki rzadko przynoszą efekty i generalnie zakładają, że optymalizacja nie jest warta wysiłku. Prawda jest taka, ze oni nie mają wystarczającego doświadczenia w pisaniu naprawdę optymalnego kodu a ich frustracja uniemożliwia wykorzystanie takiego doświadczenia Ostatnie części tego rozdziału poświęcono zademonstrowaniu co można osiągnąć kiedy optymalizujemy program. Pamiętaj zawsze o tym przykładzie, kiedy odczuwasz frustrację i jako początkujący wierzysz ,że nie można poprawić wydajności swojego programu

*”Poprawianie implementacji algorytmu”