© Rzeszów University Press
ISSN 2719-6550
e-ISSN 2719-7417

"Journal of Education, Technology and Computer Science"
No. 7(37)/2026, Special number, Part 2
https://journals.ur.edu.pl/jetacomps

Received: 23.05.2026
Accepted for printing: 8.07.2026
Published: 3.08.2026

DOI: 10.15584/jetacomps.spec.2026.2.12
Overviev

PAULINA WOŹNIAK-CHOJNACKA[1],
HUBERT KLEMENTOWSKI[2],

Złudzenie postępu – dlaczego wzrost mocy obliczeniowej nie przekłada się na odczuwalną wydajność oprogramowania

The Illusion of Progress – Why Increased Computing Power Does Not Translate Into Noticeable Improvements in Software Performance

[1] ORCID: 0000-0003-2650-7750, mgr, Uniwersytet Zielonogórski, Wydział Nauk Społecznych (Instytut Pedagogiki – Zakład Mediów i Technologii Informacyjnych); email: P.Wozniak-Chojnacka@kmti.uz.zgora.pl
[2] Student, Uniwersytet Zielonogórski, Wydział Nauk Społecznych; email: 112083@stud.uz.zgora.pl

Streszczenie

Rozwój technologii komputerowych powinien prowadzić do systematycznego wzrostu wydajności oprogramowania oraz poprawy doświadczeń użytkowników. W praktyce jednak często obserwowane jest zjawisko, które można określić mianem *złudzenie postępu* – mimo wielokrotnego wzrostu mocy obliczeniowej współczesnych komputerów ich użytkownicy często nie odczuwają proporcjonalnej poprawy szybkości działania systemów operacyjnych, aplikacji oraz gier komputerowych. Celem artykułu jest analiza przyczyn takiej sytuacji oraz identyfikacja czynników odpowiedzialnych za rosnącą rozbieżność między możliwościami sprzętu a rzeczywistą wydajnością oprogramowania.

W artykule omówiono wpływ wzrostu złożoności systemów informatycznych, wielowarstwowych architektur programistycznych, technologii webowych oraz zmian priorytetów współczesnej inżynierii oprogramowania. Uwagę poświęcono zjawisku *feature creep*, prawu Moore'a, prawu Wirtha oraz konsekwencjom ograniczenia nakładów na optymalizację kodu na rzecz szybszego wdrażania nowych funkcji. Jako studium przypadku przedstawiono współczesne przeglądarki internetowe, gry komputerowe oraz silnik Unreal Engine 5, których wymagania sprzętowe często rosną szybciej niż wydajność popularnych kart graficznych.

W zakończeniu podjęto próbę odpowiedzi na pytanie, czy obserwowany trend może ulec odwróceniu w związku z rosnącymi kosztami sprzętu komputerowego, spowolnieniem wzro-

stu wydajności układów scalonych oraz zwiększonym zapotrzebowaniem na zasoby obliczeniowe generowanym przez rozwój sztucznej inteligencji. Wyniki analizy wskazują, że przyszłość wydajności oprogramowania będzie zależała nie tylko od dalszego rozwoju sprzętu, ale również od ponownego uznania optymalizacji za jeden z kluczowych celów inżynierii oprogramowania.

Słowa kluczowe: pedagogika Montessori, edukacja alternatywna, samodzielność dziecka, indywidualizacja nauczania, rozwój dziecka

Abstract

Advances in computer technology should lead to a steady increase in software performance and an improvement in the user's experience. In practice, however, a phenomenon is often observed that can be described as the 'illusion of progress' – despite the manifold increase in the computing power of modern computers, their users often do not perceive a proportional improvement in the speed of operating systems, applications and computer games. The aim of this article is to analyze the causes of this situation and to identify the factors responsible for the growing disparity between hardware capabilities and actual software performance.

The article discusses the impact of the growing complexity of IT systems, multi-layered software architectures, internet technologies, and shifting priorities in modern software engineering. It highlights the phenomenon of feature creep, Moore's Law, Wirth's Law, and the consequences of reducing investment in code optimization in favor of faster implementation of new features. Modern web browsers, computer games and the Unreal Engine 5 are presented as case studies, as their hardware requirements often grow faster than the performance of popular graphics cards.

In conclusion, an attempt was made to address the question of whether the observed trend might be reversed in light of rising hardware costs, a slowdown in the growth of integrated circuit performance, and the increased demand for computing resources driven by the development of artificial intelligence. The results of the analysis suggest that the future of software performance will depend not only on further hardware development, but also on a renewed focus on optimization as one of the key objectives of software engineering.

Keywords: the illusion of progress, software performance, Wirth's law, optimization, computing power, software engineering

Wstęp

Analizując ostatnie dekady, można zauważyć imponujące tempo wzrostu mocy obliczeniowej komputerów. Prawo Moore'a, które stało się dominującą regułą w informatyce, głosi, że liczba tranzystorów w mikroprocesorze podwaja się co około 2 lata. Oznacza to, że wydajność mikroprocesora również wzrośnie. Wykładniczy postęp, który opisuje to prawo, przekształcił pierwsze prymitywne komputery domowe z lat 70. w zaawansowane maszyny lat 80. i 90. XX w. Następnie dał początek szybkiemu internetowi, smartfonom oraz łączącym się z siecią samochodom, lodówkom i termostatom, które dziś stają się powszechne w użyciu. Producenci chipów celowo podążali ścieżką prawa Moore'a. Na każdym etapie twórcy oprogramowania tworzyli aplikacje, które przekraczały możliwości istniejących chipów, konsumenci oczekiwali więcej od swoich urządzeń, a producenci spieszyli się, aby sprostać temu zapotrzebowaniu, tworząc chipy

nowej generacji (Waldrop, 2016). W praktyce oznaczało to systematyczny wzrost mocy obliczeniowej komputerów przy spadających kosztach produkcji. Pomimo tego, iż współczesne procesory wykonują miliardy operacji na sekundę i pamięć operacyjna liczona jest w dziesiątkach gigabajtów, a nośniki SSD zapewniają niemal natychmiast dostęp do potrzebnych danych, co powinno sprawiać, że korzystanie z oprogramowania będzie stawało się coraz szybsze i bardziej komfortowe, wielu użytkowników ma odmienne odczucia. W rzeczywistości aplikacje uruchamiają się wolno, strony internetowe zużywają ogromne ilości pamięci, zaś nowe wersje programów często nie działają sprawniej od swoich odpowiedników sprzed lat.

Wirth zauważył, że programiści często wykorzystują dodatkową moc komputerów nie do zwiększania szybkości działania aplikacji, lecz do tego, by dodawać kolejne warstwy funkcjonalności, bibliotek i efektów wizualnych. W efekcie tego współczesny użytkownik komputera nie odczuwa proporcjonalnego wzrostu wydajności. Jak stwierdził Reiser (za: Wirth, 1995, s. 1): *Oprogramowanie staje się wolniejsze szybciej, niż sprzęt staje się szybszy*. Zatem komputer, który posiada procesor wielokrotnie szybszy o tego sprzed chociażby 15 lat, nie uruchamia podstawowych aplikacji wielokrotnie szybciej. Jego zasoby są zaś konsumowane przez coraz bardziej rozbudowane programy. Jest to spowodowane tym, iż rozwój oprogramowania podąża inną ścieżką niż rozwój sprzętu. Twórcy aplikacji wykorzystują dostępne zasoby do dodawania nowych funkcji, rozbudowy interfejsów i zwiększania warstwy abstrakcji, często kosztem efektywności działania. W rezultacie rosnące możliwości komputerów nie przekładają się proporcjonalnie na wzrost szybkości wykonywania codziennych zadań. Zjawisko polegające na stopniowym dodawaniu nowych funkcji do programu, często bez rzeczywistej potrzeby użytkowników, nazywane jest *feature creep* (rozrost funkcjonalności). Aplikacja, która miała realizować jedno zadanie skutecznie i szybko, z czasem poprzez pojawiające się nowe moduły czy rozbudowane mechanizmy staje się wolna, zwiększa obciążenie procesora, a także wzrasta jej rozmiar, zużywa więcej pamięci RAM oraz posiada większą liczb błędów. Wiele jej funkcji pozostaje używanych przez niewielki odsetek użytkowników, jednak ich obecność wpływa na działanie programu dla wszystkich. Nawet najlepsi programiści i producenci wpadają w tę pułapkę, przez co wiele popularnych aplikacji i urządzeń stale otrzymuje funkcje, których nikt nie chce (Padilla-Rodriguez, 2024).

Zatem pomimo wzrostu mocy obliczeniowej wydajność oprogramowania nie wzrasta. Użytkownik nie odczuwa zwiększenia szybkości działania aplikacji. Takie zjawisko można śmiało określić mianem *złudzenie postępu*. Prowadzi ono do sytuacji, w której konsument musi pozbywać się sprzętu, gdyż pogarszająca się efektywność oprogramowania czyni go „przestarzałym", i kupować nowy, który z tego powodu wcale nie musi być szybszy w praktyce.

Przyczyny problemu

Problem z efektywnością wykorzystywania zasobów zaczyna się już na etapie tworzenia oprogramowania. Wiele pozornie nieznaczących zaniedbań składa się w całość, tworząc produkt końcowy wypełniony nieoptymalnymi funkcjami, zbędnym kodem oraz wieloma warstwami abstrakcji. Wszystko to w celu ułatwienia procesu tworzenia oprogramowania. Rozpoczyna się to niepozornie – programista pracujący nad wieloletnim projektem niemal codziennie natrafia na jawnie nieefektywne, niepotrzebne lub wręcz absurdalne fragmenty kodu. Decyduje się jednak nic z tym nie robić, gdyż stabilność kodu ceniona jest ponad wydajność. Lepiej „nie dotykać" takiego kodu, niż go ulepszyć i ryzykować powstanie błędów – niekiedy niewidocznych w testach, lecz ukrytych pośród plątaniny zależności i połączeń znanych tylko kilku osobom z grupy kilkuset uczestników projektu. Obecna kultura programowania ceni widoczne efekty i gotową funkcjonalność uzyskaną w jak najkrótszym czasie ponad optymalność, efektywność i perfekcję. Ważne jest to, czy aplikacja spełnia określone wymagania funkcjonalne, a nie jak dobrze je spełnia. Gdy przeciętny komputer osobisty staje się coraz bardziej wydajny, coraz gorszy kod zaczyna te wymagania spełniać. Wydajność zalicza się do wymagań niefunkcjonalnych, które są notorycznie pomijane w procesie inżynierii oprogramowania. Dzieje się tak szczególnie w zwinnych metodach zarządzania projektem (*agile software development*), gdzie często brakuje narzędzi do kontrolowania i korekcji wymagań niefunkcjonalnych na etapie programowania. Zazwyczaj definiowane są w początkowych stadiach projektu, a później zapominane aż do momentu, gdy jest już zbyt późno, aby cokolwiek zmieniać (Viviani, Guerra, Melegati, Wang, 2023).

Zaniedbania wydajności wynikają również z trendów, które stały się już normą przy tworzeniu oprogramowania. Pierwotnie programowanie było silnie połączone ze sprzętem. Programista musiał rozumieć działanie procesora i innych urządzeń, aby tworzyć kod zgodny z ich architekturą oraz zdatny do uruchomienia na bardzo ograniczonych zasobach. Lata 90. przyniosły zmianę w tym podejściu – powstanie języka Java zapoczątkowało popularyzację niemalże całkowitego uniezależniania oprogramowania od sprzętu (Tiwari, 2026), skutkującego m.in. automatyzacją zarządzania pamięcią i ukrywaniem szczegółów działania pod warstwami środowiska uruchomieniowego i abstrakcji. Współczesny programista, pozbawiony ciasnych ograniczeń sprzętowych, nie musi już rozumieć szczegółów działania sprzętu, przez co tworzony przez niego kod jest często nieoptymalny dla wykonującej go jednostki centralnej, powstanie języka Java zapoczątkowało bowiem także trend, a następnie dominację paradygmatu programowania obiektowego – zastępując tym samym klasyczne podejście proceduralne (Tiwari, 2026). Obiektowe języki programowania są najlepiej widocznym symptomem wymienionych problemów. Ich popularność jest obecnie znacząca – dane z ankiety przeprowadzonej przez Stack Overflow po-

kazują procent respondentów, którzy w ciągu danego roku intensywnie pracowali w danym języku. Po ograniczeniu rezultatów do języków programowania oraz respondentów do zawodowych programistów do pięciu najpopularniejszych języków należą: JavaScript (68,8%), Python (54,8%), TypeScript (48,8%), C# (29,9%) i Java (29,6%) (Stack Overflow, 2025, 2026), z czego za języki oparte w znacznym stopniu na obiektowości można uznać Pythona, C# i Javę. Należy też zauważyć, że wysoka popularność JavaScript i TypeScript, niebędących językami typowo obiektowymi, wynika głównie z dominującej pozycji technologii webowych, w których wykorzystanie jednego z nich po stronie frontendu jest często koniecznością, a nie świadomym wyborem określonego paradygmatu. W tym kontekście popularność języków obiektowych widoczna jest głównie po stronie backendu – wśród najpopularniejszych frameworków backendowych dominują Spring Boot (Java), ASP.NET Core (C#) i Django (Python) (Stack Overflow, 2025, 2026). Jedną z przyczyn popularności programowania obiektowego jest możliwość organizacji kodu w postaci współpracujących obiektów, co ułatwia modularyzację, utrzymanie oraz rozwój większych projektów. Kod, który jest uporządkowany dla człowieka, często jednak taki nie jest „z perspektywy" procesora.

Już na etapie samej koncepcji podejścia obiektowego, czyli „organizacji kodu wokół danych", pojawia się potencjalne napięcie pomiędzy abstrakcją programu a wymaganiami nowoczesnych procesorów dotyczącymi zarządzania danymi, w rzeczywistości języki obiektowe nie organizują bowiem kodu wokół fizycznych układów pól w pamięci, lecz jedynie tworzą logiczne asocjacje w ramach poszczególnych typów danych, jednocześnie ukrywając ich szczegóły pod warstwami abstrakcji. Wraz ze wzrostem poziomu skomplikowania aplikacji staje się to coraz większym problemem. Gdy klasy mają skomplikowaną sieć zależności między sobą, powstaje tendencja do częstych skoków do odległych od siebie obszarów pamięci. Współczesne procesory wykorzystujące hierarchię pamięci cache działają efektywnie przy sekwencyjnym i przewidywalnym dostępie do danych (Drepper, 2026). Klasyczne podejście do obiektowości często nie spełnia tych warunków – obiekty mogą być rozproszone po stercie, operacje na nich mogą generować nieregularne wzorce dostępu do pamięci. Ta słaba lokalność danych nieraz powoduje znacząco niższą wydajność niż w innych rozwiązaniach projektowanych z uwzględnieniem lokalności danych (np. *data-oriented design)* (Fabian, 2026). Problemy z optymalnym działaniem procesora pogłębiają się przy stosowaniu mechanizmów wymagających dynamicznego wiązania wywołań funkcji (np. polimorfizm). Wprowadza ono problemy w jednoznacznym określeniu zachowania programu podczas kompilacji, przez co część pracy, którą normalnie wykonałby kompilator, musi zostać zrealizowana w momencie wykonywania dynamicznie wiązanej funkcji, co zabiera cenne zasoby, głównie poprzez większą ilość operacji dostępu do pamięci. Jednocze-

śnie ta niejasność sposobu wykonywania kodu może – w zależności od sytuacji i kompilatora – uniemożliwiać optymalizację wydajności wykonywania kodu poprzez wstawianie kodu funkcji w miejscu jej wywołania w momencie kompilacji (mechanizm *inline*). Testy porównawcze pokazują, że wywołania funkcji wirtualnych mogą być od dwóch do nawet dziesięciu razy wolniejsze niż wywołania bezpośrednie – jest to jednak silnie zależne od konkretnego przypadku użycia (Wagh, 2026).

Charakterystyczna dla większości języków obiektowych separacja oprogramowania od sprzętu objawia się również w konieczności automatycznego zarządzania nieużywaną pamięcią (*garbage collector* – GC). Wymaga to detekcji nieużywanych obiektów w trakcie działania programu oraz odpowiedniego zarządzania nimi, co stanowi znaczący narzut podczas *runtime*. Badania nad produkcyjnymi GC w OpenJDK 17 pokazują, że pochłaniają one co najmniej 7–82% dodatkowego czasu zegarowego i 6–92% dodatkowych cykli CPU w porównaniu do hipotetycznego braku GC. Nowsze typy GC mające na celu unikanie generowania dużych opóźnień są znacznie bardziej kosztowne niż prostsze GC typu *stop-the-world* (Cai, Blackburn, Bond, Maas, 2026), co paradoksalnie może oznaczać stosowanie mniej wydajnych GC w sytuacjach, w których czas i termin wykonania operacji mają znaczenie.

O ile widoczne jest, że popularyzacja programowania obiektowego zapoczątkowała trend poświęcania wydajności w imię prostoty i łatwości tworzenia kodu, o tyle obecnie zależność ta jest kontynuowana. Objawia się to w znaczących zmianach w wyborze języków programowania, widocznych w dużej i rosnącej popularności języków skryptowych – głównie Python i JavaScript/TypeScript. Powracając do przytoczonych wcześniej statystyk Stack Overflow (2025, 2026), można zauważyć przede wszystkim wysoką (54,8%) i stale rosnącą – aż o 7% więcej w 2025 niż w 2024 r. – pozycję języka Python. Jest to znaczące w kontekście pogarszającej się wydajności, gdyż Python należy do jednych z mniej wydajnych języków programowania – w dużej mierze z powodu bycia językiem interpretowanym oraz dynamicznie typowanym. Zamiana kodu Pythona na kod maszynowy na bieżąco podczas wykonywania programu jest znaczącym, dodatkowym zużyciem zasobów procesora. Interpretery tworzące reprezentację pośrednią kodu przed uruchomieniem są tutaj mniej problematyczne, ale nadal nie są w stanie dorównać wydajności języków kompilowanych. Problem pogłębia się przy dynamicznym typowaniu zmiennych, gdzie interpreter musi dokonać walidacji typu zmiennej zazwyczaj przy każdym jej wywołaniu. Zwyczajne wywołanie funkcji może więc oznaczać nawet kilkanaście operacji sprawdzania typu – jeszcze zanim sama funkcja zacznie się wywoływać. Badania porównawcze wydajności GCC (C++), OpenJDK (Java), Go, V8 (Node.js) oraz CPython (Python) zaprezentowane na konferencji USENIX pokazują, że w porównaniu z GCC, kod uruchamiany w CPython wykonuje wymagające obliczeniowo za-

dania 29,5 raza wolniej, a V8 8,01 raza wolniej. Przewaga V8 wynika głównie z obecności mechanizmu *just-in-time compiler*, który powoduje, iż JavaScript uruchamiany na V8 nie jest już językiem w 100% interpretowanym (Lion, Chiu, Stumm, 2026). Duża popularność JavaScript i TypeScript (68,8% i 48,8%) (Stack Overflow 2025, 2026) powoduje rozprzestrzenianie się ich poza swoje standardowe, niemożliwe do uniknięcia zastosowania. Dużą popularnością cieszą się obecnie takie frameworki, jak m.in. Node.js (40,7%), Next.js (18,6%), Express (18,2%) (Stack Overflow 2024, 2026), umożliwiające tworzenie full-stackowych aplikacji opartych wyłącznie na JS/TS. Coraz częstsze staje się także wykorzystywanie tych języków do aplikacji desktopowych poprzez takie frameworki, jak Electron (npm trends, 2026). Z tych powodów trend korzystania z języków skryptowych może mieć w przyszłości negatywny wpływ na wydajność większości typów oprogramowania.

Wszystkie opisane poprzednio zmiany w tworzeniu oprogramowania na przestrzeni lat mają ostatecznie na celu ułatwić szybkie tworzenie skomplikowanej funkcjonalności. Takie stawianie wymagań funkcjonalnych ponad niefunkcjonalne najlepiej widać, patrząc właśnie na technologie, których głównym celem jest pomijanie części pracy potrzebnej do implementacji danej funkcjonalności. Mowa tutaj o wszelkiego rodzaju frameworkach i bibliotekach, które oferują wiele gotowych rozwiązań, eliminując potrzebę jej ręcznej implementacji za każdy razem i w każdym projekcie. Są to mechanizmy niezbędne do funkcjonowania rynku informatycznego, lecz tak jak każde inne, mogą być nadużywane. Niestety, obecnie częste jest podejście kolokwialnie określane jako „rzucanie technologiami w problemy", czyli dodawanie niepotrzebnie wielu rozbudowanych bibliotek i frameworków do każdego, nawet mało skomplikowanego projektu. Takie praktyki skutkują często znacznie wyższym zużyciem pamięci operacyjnej oraz nadmiernie rozbudowanymi implementacjami prostych funkcjonalności, pogarszającymi tym samym wydajność.

Przykładem najlepiej pokazującym ten problem jest Electron – framework umożliwiający tworzenie aplikacji desktopowych przy użyciu technologii webowych: JavaScript, HTML i CSS. Jego główną zaletą jest możliwość napisania jednego kodu działającego na wielu systemach operacyjnych bez potrzeby znajomości natywnych platform (Build cross-platform, 2026). Cenę wygody z tego wynikającej ponoszą użytkownicy: Electron osiąga swój cel przez dołączenie do każdej aplikacji pełnej, osobnej kopii silnika przeglądarki Chromium wraz ze środowiskiem Node.js (Electron: Prerequisites, 2026). Oznacza to, że użytkownik mający jednocześnie otwartych kilka aplikacji bazujących na Electron uruchamia trzy oddzielne instancje przeglądarki Chrome, każdą z własnym stosem procesów, własną pamięcią i własnym silnikiem renderowania (Electron: Process Model, 2026). Skutkiem tak gigantycznego narzutu jest znacznie gorsza wydajność i zużycie RAM. W porównaniu z aplikacją Flutter (alternatywnym

frameworkiem wieloplatformowym), prosty program na platformie macOS wyświetlający „Hello, World!" zużywa aż 99,6 MB, gdy jest napisany w Electron, oraz 37,9 MB, gdy jest napisany w Flutter. W przypadku programu wyświetlającego animacje Electron używa 215% zasobów CPU i 41% GPU, zaś Flutter na tym samym urządzeniu wykorzystuje 130,6% CPU oraz 12,8% GPU. Jednocześnie Electron używa aż 2261,3 MB pamięci RAM, podczas gdy Flutter wykorzystuje 168,5 MB (macOS Performance, 2026). Widoczne tutaj różnice w wydajności są znaczące szczególnie dlatego, iż stosowanie Electron jest jedynie kwestią optymalizacji procesu tworzenia oprogramowania, która nie zmienia nic dla użytkownika końcowego pod względem funkcjonalności. Electron pokazuje więc, że nadmiarowy stos technologiczny projektu nie musi wcale oznaczać zbyt wielu technologii – nawet jedna, źle dobrana, może bowiem znacząco pogorszyć wydajność i niepotrzebnie skomplikować projekt.

Przykłady zjawiska

Jednym z najlepszych przykładów omawianego problemu są współczesne przeglądarki internetowe, głównie te oparte na silniku Chromium. Wykorzystują one architekturę wieloprocesową, która zwiększa bezpieczeństwo i stabilność działania, lecz jednocześnie prowadzi do większego zużycia pamięci operacyjnej RAM (*The Chromium Projects*…). Wiąże się to z faktem, iż przeglądarki uruchamiają każdą kartę w osobnym procesie, wykonują skomplikowany kod JavaScript, obsługują aplikacje internetowe przypominające programy desktopowe i stale komunikują się z usługami sieciowymi. Otworzenie kilkunastu kart może skutkować wykorzystaniem nawet kilku gigabajtów pamięci RAM. Ponadto występują problemy z płynnością, gdyż mimo ogromnego wzrostu wydajności procesorów użytkownicy wciąż obserwują przycinanie interfejsu, opóźnienia podczas przełączania kart, spowolnienia po dłuższym czasie pracy czy wysokie zużycie energii na laptopach.

Podobne problemy można zauważyć w przypadku systemów operacyjnych. Każda kolejna generacja oferuje nowe funkcje i atrakcyjniejszy interfejs, ale jednocześnie zwiększa wymagania sprzętowe. Komputer uznawany za wydajny kilka lat wcześniej może działać odczuwalnie wolniej po aktualizacji systemu. Wart przeanalizowania jest problem wydajności na przykładzie komputera z 2010 r. z systemem Windows 7 oraz komputera z 2025 r. z systemem Windows 11. Użytkownik tutaj nie odczuwa wzrostu wydajności odpowiadającego wzrostowi mocy sprzętu. Otwarcie przeglądarki w przypadku obu systemów trwa podobnie, zaś eksplorator plików czasami reaguje wolniej, menu start potrzebuje więcej zasobów, wiele aplikacji zajmuje setki MB pamięci w komputerach z systemem Windows 11. Jedną z przyczyn jest nadmiar warstw programowych. Windows 7 był w większości systemem natywnym, Windows 11 zawiera zaś liczne usługi działające w tle, telemetrię, synchronizację z chmurą,

zabezpieczenia działające w czasie rzeczywistym oraz komponenty webowe. Każda warstwa pochłania część dostępnej mocy. Drugą przyczyną jest technologia Microsoft Edge WebView2 i „strony internetowe udające aplikacje". Wiele elementów Windows 11 jest obecnie renderowanych podobnie jak strony internetowe. Takie elementy, jak widgety, część ustawień, aplikacje systemowe, komunikatory, programy firm trzecich, są wygodne dla programistów, ale mniej efektywne niż kod natywny. Tu zauważalne jest omówione wcześniej prawo Wirtha. Programiści rzadziej optymalizują kod, zakładają bowiem dostępność coraz mocniejszego sprzętu, a nowe funkcje pochłaniają zyski wynikające z rozwoju procesorów (TechnoDigital, 2026).

Najbardziej spektakularny okazuje się przykład współczesnych gier komputerowych. Rozwój technologii komputerowej powinien prowadzić do coraz płynniejszego działania gier, co wydawałoby się na pierwszy rzut oka oczywiste. W rzeczywistości wielu graczy doświadcza sytuacji wręcz odwrotnej. Zakup nowego sprzętu nie daje oczekiwanych efektów, a gracze nadal muszą obniżać ustawienia graficzne lub korzystać z technik skalowania obrazu. Powstaje wrażenie, że postęp sprzętowy nie przynosi oczekiwanych korzyści. Szczególnie wyraźnie widać to na przykładzie Unreal Engine 5. Nowoczesne tanie GPU dorównują najlepszym sprzed kilkunastu lat, lecz nie są w stanie udźwignąć nowych gier. Silnik Unreal Engine 5 oferuje bardzo szczegółowe modele 3D oraz zaawansowane efekty świetlne i cieniowanie. Ponadto udostępnia zwirtualizowany system geometrii zwany Nanite, który wykorzystuje wewnętrzny format siatki i technologię renderowania do renderowania szczegółów w skali pikseli i dużej liczby obiektów (*Przegląd geometrii*…), oraz Lumen, czyli globalne oświetlenie w czasie rzeczywistym (*Globalne oświetlenie i odbicia*…). Dzięki temu twórcy mogą tworzyć światy o niespotykanym wcześniej poziomie szczegółowości. Technologie te wymagają jednak ogromnej mocy obliczeniowej, stąd w praktyce wiele gier opartych na Unreal Engine 5 ma trudności z utrzymaniem stabilnej płynności nawet na nowoczesnych komputerach. Wskazuje na to również szef Epic Games, który podkreśla, że deweloperzy za bardzo skupiają się na najmocniejszych konfiguracjach sprzętowych, a testowanie na słabszych maszynach zostawiają na sam koniec (Woźnicki, 2025). Przykład współczesnych gier pokazuje, że postęp technologiczny nie zawsze przekłada się na odczuwalny wzrost wydajności, wzrost mocy obliczeniowej kart graficznych jest bowiem bardzo często neutralizowany przez coraz bardziej wymagające silniki, rozbudowane efekty graficzne oraz niedostateczną optymalizację. W rezultacie użytkownicy gier mogą błędnie postrzegać nowe układy graficzne jako niewystarczająco wydajne, mimo że pod względem czystej mocy obliczeniowej przewyższają one dawne konstrukcje z najwyższej półki. Tak absurdalne wymagania sprzętowe sprawiają, iż przez złą optymalizację gracze nie wiedzą, że ich tanie nowe GPU jest na poziomie najlepszych GPU sprzed 10 lat. To właśnie istota *złudzenia postępu* w świecie współczesnych gier komputerowych.

Podsumowanie

Pomimo ogromnego postępu technologicznego, jaki przewidywało prawo Moore'a, użytkownicy nie zawsze odczuwają proporcjonalny wzrost szybkości działania komputerów. Wyjaśnia to dość jasno prawo Wirtha, które wskazuje, iż oprogramowanie rozwija się w sposób pochłaniający coraz większą część dostępnych zasobów sprzętowych. Głównymi przyczynami takiego zjawiska są rozrost funkcjonalności (*feature creep*), rozrost kodu aplikacji, wykorzystywanie coraz większej liczby bibliotek i frameworków, wzrost wymagań współczesnych usług internetowych oraz większe oczekiwania dotyczące grafiki i funkcjonalności. Rezultatem tego jest fakt, iż dodatkowa moc obliczeniowa komputerów jest często konsumowana przez coraz bardziej złożone oprogramowanie. W konsekwencji użytkownik nie odczuwa tak dużej poprawy wydajności, jakiej można by oczekiwać na podstawie parametrów sprzętu. Kluczowym wyzwaniem współczesnej informatyki staje się więc nie tylko tworzenie szybszego sprzętu, ale również projektowanie bardziej efektywnego i oszczędnego oprogramowania. Programiści powinni zatem większą uwagę poświęcać analizie wydajności, usuwaniu zbędnego kodu oraz ograniczaniu liczby zależności i bibliotek. Tworzone przez nich aplikacje powinny koncentrować się na najważniejszych funkcjach, zamiast nieustannie zwiększać zakres możliwości. Współczesna inżynieria oprogramowania częściowo zmieniła swoje priorytety z efektywności działania na szybkość rozwoju i wygodę produkcji. Efektem jest sytuacja, w której wzrost mocy sprzętu nie przynosi użytkownikom wyraźnych korzyści. Z perspektywy konsumenta prowadzi to do poczucia, że sprawny technicznie sprzęt staje się „przestarzały" nie z powodu własnych ograniczeń, lecz dlatego, że nowe wersje oprogramowania wymagają coraz większych zasobów, nie oferując proporcjonalnego wzrostu funkcjonalności czy jakości. To jeden z najbardziej charakterystycznych przejawów współczesnego „złudzenia postępu" w informatyce.

Biorąc pod uwagę rozwój AI oraz rosnące koszty zaawansowanych podzespołów można przypuszczać, że wkrótce zakończy się okres, w którym nieefektywność oprogramowania była ukrywana przez stale taniejącą i coraz wydajniejszą elektronikę. Jeżeli moc obliczeniowa przestanie rosnąć tak szybko jak dotychczas, a ceny sprzętu pozostaną wysokie, producenci mogą zostać zmuszeni do ponownego uznania optymalizacji za jeden z głównych celów inżynierii oprogramowania. Taka sytuacja nie jest jednak przesądzona, ponieważ te same technologie AI mogą jednocześnie sprzyjać dalszemu wzrostowi złożoności oprogramowania.

Literatura

Build cross-platform desktop apps with JavaScript, HTML, and CSS. Pobrano z: https://electronjs.org/ (4.06.2026).

Cai, Z., Blackburn, S., Bond, M., Maas, M. (2022). *Distilling the Real Cost of Production Garbage Collectors*. Pobrano z: https://arxiv.org/pdf/2112.07880 (1.06.2026).

Drepper, U. (2007). *What Every Programmer Should Know About Memory*. Pobrano z: https://www.akkadia.org/drepper/cpumemory.pdf (1.06.2026).

Electron: *Prerequisites*. Pobrano z: https://www.electronjs.org/docs/latest/tutorial/tutorial-prerequisites (4.06.2026).

Electron: *Process Model*. Pobrano z: https://www.electronjs.org/docs/latest/tutorial/process-model (4.06.2026).

Fabian, R. (2018). *Data-Oriented Design*. Pobrano z: https://www.dataorienteddesign.com/dodbook/ (1.06.2026).

Globalne oświetlenie i odbicia Lumen. Pobrano z: https://dev.epicgames.com/documentation/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine (4.06.2026).

Lion, D., Chiu, A., Stumm, M. (2022). *Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?* Pobrano z: https://www.usenix.org/system/files/atc22-lion.pdf (3.06.2026).

macOS Performance Comparison: Flutter Desktop vs. Electron. Pobrano z: https://getstream.io/blog/flutter-desktop-vs-electron/ (4.06.2026).

npm trends: electron. Pobrano z: https://npmtrends.com/ (4.06.2026).

Padilla-Rodriguez, M. (2024). *How "Feature Creep" Is Ruining Software, Gadgets, and Video Games*. Pobrano z: https://www.howtogeek.com/how-feature-creep-is-ruining-software-gadgets-and-video-games/ (4.06.2026).

Przegląd geometrii wirtualizowanej nanitów. Pobrano z: https://dev.epicgames.com/documentation/unreal-engine/nanite-virtualized-geometry-in-unreal-engine (4.06.2026).

Stack Overflow 2024 Developer Survey. Pobrano z: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-webframe-prof (4.06.2026).

Stack Overflow 2025 Developer Survey. Pobrano z: https://www.wscubetech.com/blog/backend--languages/ (2.06.2026).

The Chromium Projects. *Multi-process architecture*. Pobrano z: https://www.chromium.org/developers/design-documents/multi-process-architecture/ (2.06.2026).

TechnoDigital (2026). *Od Windows XP do Windows 11: ostateczne porównanie wydajności.* Pobrano z: https://informatecdigital.com/pl/Od-Windows-XP-do-Windows-11%3A-najlepsze--por%C3%B3wnanie-wydajno%C5%9Bci/ (4.06.2026).

Tiwari, P. (2024). *The Evolution of Object-Oriented Programming: From Simula to the Modern Era*. Pobrano z: https://mptmt16.medium.com/the-evolution-of-object-oriented-programming--from-simula-to-the-modern-era-cd7096471eb8 (3.06.2026).

Viviani, L., Guerra, E., Melegati, J., Wang, X. (2023). *An Empirical Study About the Instability and Uncertainty of Non-functional Requirements*. W: C.J. Stettina, J. Garbajosa, P. Kruchten, (red.), *Agile Processes in Software Engineering and Extreme Programming* (s. 77–93). Cham: Springer. https://doi.org/10.1007/978-3-031-33976-9_6

Wagh, P. (2025). *The Real Cost of Virtual Functions: A Performance Deep Dive*. Pobrano z: https://www.linkedin.com/pulse/real-cost-virtual-functions-performance-deep-dive-pawan-wagh-uwdlc (2.06.2026).

Waldrop, M. (2023). The chips are down for Moore's law. *Nature*, *530*, 144–147. https://doi.org/10.1038/530144a

Wirth, N. (1995). A Plea for Lean Software. *IEEE Computer*, *28*(2), 64–68. Pobrano z: https://github.com/sysprv/demo-corporate-documentation-public/blob/master/Niklaus%20Wirth%20-%20A%20Plea%20for%20Lean%20Software.pdf (1.06.2026).

Woźnicki, Z. (2025). *Tim Sweeney uważa, że słaba optymalizacja gier na Unreal Engine 5 to wina deweloperów*. Pobrano z: https://www.gry-online.pl/newsroom/tim-sweeney-uwaza-ze-slaba-optymalizacja-gier-na-unreal-engine-5/zc2ea40 (4.06.2026).