## A Turk, a Turk, a Turk

*Christopher Tayler is a contributing editor at the LRB.*

Snow by Orhan Pamuk, translated by Maureen Freely.

Faber, 436 pp., £12.99, May 2004 lrb · 5 August 2004

‘Be yourself,’ a beautiful woman called Ipek says to Ka, the protagonist of Orhan Pamuk’s newly translated novel, Snow (Kar, 2002), when he asks how to win her heart. Though kindly meant, it’s discouraging advice to give one of Pamuk’s characters, for whom being themselves is difficult. ‘No one can ever be himself in this land,’ says the shadowy figure who may or may not be responsible for the double murder that closes The Black Book (Kara Kitap, 1990; translated in 1994). ‘In the land of the defeated and oppressed, to be is to be someone else. I am someone else; therefore I am.’

The land he’s talking about is Turkey, where, many of Pamuk’s characters believe, an authentic cultural identity is particularly hard to come by. Sometimes the country’s mixed heritage is seen as cause for celebration: Enishte Effendi, the hero’s uncle in My Name Is Red (Benim Adim Kirmizi, 1998; translated in 2001), makes a long speech praising the cross-cultural borrowings of Ottoman miniaturists. ‘“To God belongs the East and the West,”’ he says, quoting the Koran. ‘May He protect us from the will of the pure and unadulterated.’ More often, however, Pamuk shows us people who fear that living, as he puts it, ‘in a Westernised fashion in a country that is essentially not Western’ has drained them of selfhood. Doubles and false identities proliferate in his books, and his characters frequently suffer from feelings of inauthenticity. They catch themselves acting like people in movies (‘I don’t know from what film I had pinched this gesture’). On occasion they feel ‘completely empty inside’. ‘It’s futile to search for ... the original of which we are all mere copies,’ someone says in The New Life (Yeni Hayat, 1994; translated in 1997). Few of the characters find this a comforting thought.

Unlike some of his characters, Pamuk doesn’t worry much about cultural purity. The New Life’s surly narrator says: ‘This newfangled plaything called the novel, which is the greatest invention of Western culture, is none of our culture’s business,’ but Pamuk takes the opposite view. His books cheerfully plunder a bewildering range of material: stories, ideas and images from Rumi, Attar and the Arabian Nights rub shoulders with borrowings from Dostoevsky, Rilke, Proust and Joyce. Like Borges, whom he also admires, he cultivates spooky parallels between Islamic mysticism and European Modernism. In a less rarefied vein, he writes well about loneliness, nostalgia, cities and weather, and his novels have interesting things to say about politics and culture in Turkey. They even offer the traditional satisfactions of character and plot – although the novels don’t always manage to do all of these things at once.

His first two books, Cevdet Bey ve Ogullari (1982) and Sessiz Ev (1983), haven’t been translated into English. Cevdet Bey ve Ogullari (‘Cevdet Bey and His Sons’) is said to be a naturalistic saga about three generations of a Turkish family: Cevdet Bey, who is based on Pamuk’s grandfather, builds the family fortune between the collapse of the Ottoman Empire and the foundation of the Turkish republic. Sessiz Ev, which has been translated into French as La Maison du silence, is said to be more technically ambitious. Set around 1980, at the time of Turkey’s military coup and the vigorous crackdown on leftists and Islamists that followed, it’s narrated in Faulknerian style by five different characters, the descendants of an Atatürk-like patriarch whose high hopes for Westernisation have congealed, in his grandchildren’s generation, into post-Kemalist malaise. One of these characters, a dissolute archivist called Faruk Darvinoglu, returns as the ‘editor’ of Pamuk’s third novel, The White Castle (Beyaz Kale, 1985), his first to be translated into English – in 1990, very readably, by Victoria Holbrook.

The White Castle takes the form of a manuscript that Faruk has uncovered and rewritten in contemporary Turkish. After a short introduction, in which Faruk announces that he’s given up claiming the story has any ‘relevance to ... East-West relations’, the novel cuts to his rendering of the 17th-century confession of an Italian scholar enslaved by Ottoman Turks. Captured at sea while travelling to Naples, the anonymous narrator ends up as the property of Hoja, an Istanbul scholar who’s curious about ‘Frankish’ ways. The two become collaborators – first on firework displays for the sultan, and then on containing an outbreak of plague that the Ottoman authorities are ill-equipped to cope with.

Increasingly, Hoja – who bears an uncanny resemblance to the narrator – becomes obsessed with the mysteries of European self-fashioning. Over the course of several years, he and the narrator study one another, gazing into mirrors and writing answers to the question ‘Why am I what I am?’ Hoja decides that the Franks’ successes derive from their greater capacity for self-scrutiny, and takes to interrogating his captive about his sins. When the sultan’s armies cross the Danube, Hoja and the narrator accompany them. Hoja arranges bizarre interrogations of the Christian villagers, who, he feels, are withholding the ‘deeper truth’ he’s looking for. In order ‘to prove what kind of men “they”, and furthermore “we”, were’, the villagers are lined up and ordered to confess their misdemeanours. The results disappoint him.

Hoja has also used his knowledge of the Frankish sciences to build a primitive tank. When this lumbering vehicle founders in the mud beneath the walls of the castle they’re unsuccessfully besieging, the soldiers accuse Hoja’s infidel slave of conspiring to bring about their defeat. Hoja takes pity on the narrator and the two men swap identities: Hoja escapes to Italy, where he writes books about his adventures among the Turks; the narrator returns to a well-funded retirement in an Ottoman villa. He misses his former master bitterly and, late in life, confesses the truth to Evliya Chelebi, a famous (real-life) traveller. Chelebi is sceptical about his story and questions the propriety of his telling it: ‘To search within, to think so long and hard about our own selves, would only make us unhappy. This is what had happened to the characters in my story: for this reason heroes could never tolerate being themselves, for this reason they always wanted to be someone else.’

Turkish readers have pointed out that Pamuk’s first two books recapitulate the shift from realism to Modernism. The White Castle equally self-consciously takes the next literary-historical step, and at times reads like an Ottoman version of, say, Calvino’s Our Ancestors. Occasionally the story flags: the characters don’t get out much, and such capitalised concepts as the Gaze and the Other often threaten to lurch onstage. But it’s done with enough brio not to degenerate into an academic exercise, and the question-mark over the narrator’s real identity is skilfully managed. So, too, is the oddly touching relationship that develops between captive and captor, whose curiosity about one another is obliquely compared to Proust’s narrator’s obsession with Albertine. Faruk, the notional editor, opens the book with an epigraph from Proust (in ‘the mistranslation of Y.K. Karaosmanoglu’): ‘To imagine that a person who intrigues us has access to a way of life unknown and all the more attractive for its mystery ... what else is this but the birth of a great passion?’

Pamuk’s next two novels, The Black Book and The New Life, are more strenuously avant-garde; they also won him enormous fame in Turkey, where The New Life notched up unprecedented sales. Both are metaphysical detective stories revolving around the narrators’ attempts to decipher mysterious texts and track down equally mysterious women. In The Black Book, Galip, an Istanbul lawyer, searches for his wife, who seems to have absconded with her brother, Jelal, an enigmatic columnist; the action takes place during the build-up to the 1980 coup. Jelal’s columns, which resemble prose poems or parables, alternate with chapters following Galip as he travels around the city impersonating Jelal. The New Life is narrated by Osman, a student who starts travelling around Turkey by bus in search of a girl who has given him a mysterious, life-changing book. He adopts a false identity and succeeds in catching up with her, but loses her again after becoming embroiled in a strange conspiracy.

Both books are atmospheric, allusive and densely packed with ideas. The Black Book includes an Ottoman version of Dostoevsky’s Grand Inquisitor, a fantasia on ‘The Day the Bosphorus Dries Up’, an allegorical excursion into ‘the morbid history of Turkish mannequin making’ and an occult sect based on Hurufism – a Sufi tradition resembling cabbalism. Both books are full of satirical images designed to irritate both ardent Kemalists and devotees of political Islam. In The New Life, Osman visits an exhibition of amazing Turkish inventions, where he sees a cuckoo clock ‘that provides the answer to the problem of the call to prayer’:

This clock automatically settled the Westernisation-versus-Islamisation question through a modern device: instead of the usual cuckoo bird, two other figures had been employed, a tiny imam who appeared on the lower balcony at the proper time for prayer to announce three times that ‘God is great!’ and a minute toy gentleman wearing a tie but no moustache who showed up in the upper balcony on the hour, asserting that ‘Happiness is being a Turk, a Turk, a Turk.’

At the same time, however, both novels are uncompromisingly difficult to read. This is only partly Pamuk’s fault. It’s true that the characters are cut-outs, and that it’s not easy to keep track of all the wilderness-of-mirrors effects: when it turns out that many of Jelal’s back numbers, as studied by Galip, are esoteric commentaries on Jalal al-Din Rumi’s Mathnawi – a 13th-century work which Robert Irwin has described as ‘a great sprawling series of stories within stories’ in which ‘the inner stories ... indicate the mystical meanings of the outer stories’ – the reader’s head starts to throb. It’s also true that Borges-style novels are usually more enticing in Borges-style summary than they are in reality. But none of this would be so much of a problem – for Anglophone readers, at least – if it weren’t for the tortured writing of Güneli Gün, who translated both novels. Gün has a wayward sense of register and her English sentences are often hard to decode. Her translations read very awkwardly in a way that all the other translations of Pamuk do not.

Pamuk was luckier with Erdag Göknar, who translated My Name Is Red, the novel that found the large international readership he deserves. The English text is elegant and clear, and this is easily Pamuk’s most accessible book, as well as his most straightforwardly entertaining. An art-historical murder mystery that takes place in 1591 among the illustrators employed by the Ottoman Sultan Murat III, it’s narrated in soliloquies by the major and minor characters, with many sudden, suspenseful shifts of point of view. The Black Book’s worries about distinguishing between ‘pure images’ and ‘the second-hand’ become much less inscrutable when transferred to a team of painters, and the frequent bouts of elaborate storytelling also seem more plausible when they’re being held in the late 16th century. Most of all, the mysterious paradoxes about identity and Westernisation work well in an intellectual thriller. The murderer, when he’s finally unmasked, turns out to be not a fanatical chauvinist (as you’re led to expect by the occasional echoes of The Name of the Rose) but a painter who has killed to protect his experiment in the Frankish art of portraiture – an experiment whose failure seems to unhinge him. Meanwhile, a vigilante mob of proto-fundamentalists lays waste the artistic redoubts of Istanbul.

Despite this gloomy ending – which may gesture towards the violence employed by Turkish Westernisers as well as their opponents since Atatürk’s day – it’s clear that Pamuk is firmly on the side of such non-violent admirers of Frankish arts as Enishte Effendi and the novel’s hero, Black. Göknar’s translation of My Name Is Red was published in 2001; after 11 September, Pamuk was in demand as a commentator. In November, he published an article in the New York Review of Books describing the likely results of an ill-judged response to the terrorist attacks. Giving vent to ‘self-righteous Western nationalism’, he said, would ‘drive the rest of the world into defiantly contending that two plus two equals five, like Dostoevsky’s Underground Man’. He also tried to explain the ‘overwhelming feeling of humiliation that is experienced by most of the world’s population’:

Today an ordinary citizen of a poor, undemocratic Muslim country, or a civil servant in a third-world country or in a former socialist republic ... is aware of how insubstantial is his share of the world’s wealth; he knows that he lives under conditions that are much harsher and more devastating than those of a ‘Westerner’ and that he is condemned to a much shorter life. At the same time, however, he senses in a corner of his mind that his poverty is to some considerable degree the fault of his own folly and inadequacy ... This is the grim, troubled private sphere that neither magical realistic novels that endow poverty and foolishness with charm nor the exoticism of popular travel literature manages to fathom. And it is while living in this private sphere that most people in the world today are afflicted by spiritual misery.

His new novel, Snow, is in part a ‘Dostoevskian political novel’ addressing the tangled problems of this ‘private sphere’. But it also retreats from the rather lofty attitude implied by Pamuk’s talk of ‘folly and inadequacy’. In his fiction, Pamuk has always been at pains to distance himself from the Turkish tradition of the ‘village novel’, in which – in Güneli Gün’s dismissive summary – ‘the author, more often than not a member of the middle-class intelligentsia, depicts the trials and tribulations of godforsaken peasants in an effort to “educate” the reading public (also composed of the middle class)’. Many of the characters in Snow come in for fairly withering scrutiny, but – as Maureen Freely, the novel’s translator, put it in her review of the Turkish edition – those who come out worst are the ‘bourgeois intellectuals who are just passing through’. These include the book’s narrator, a novelist called ‘Orhan’ who, during his infrequent appearances onstage, repeatedly expresses his puzzlement when people don’t pay sufficient attention to his literary fame. He also drinks quite a lot.

The protagonist, Ka (a pseudonym – his real name is Kerim Alakusoglu), has more conventional problems with his identity than most of Pamuk’s leading men. He is a poet and a deracinated bourgeois intellectual; he has only once visited a mosque, during his childhood – when his parents’ maid took him. He has read a lot of European literature, knows nothing of serious poverty and was peripherally involved with the student left during the late 1970s. This led to 12 years of political exile in Frankfurt after the 1980 coup, and has left him disillusioned with politics. The novel begins in the early 1990s. Ka has returned to Turkey to attend his mother’s funeral. A friend who writes for the Republican has sent him to Kars, a run-down city near the Armenian border, to report on the local elections and the recent ‘suicide epidemic’ among the city’s young women. The narrator – who doesn’t reveal his identity until much later – says that he’s an old friend of Ka’s and hints that ‘our traveller’ is going to have a bad time.

Ka checks in at the Snow Palace Hotel and introduces himself to the local police chief. Kars is not a tranquil place. The Islamists are poised to win the municipal elections, Kurdish guerrillas are operating in the area, and the secret police apparently employ most of the local population as informers. Ka trudges through the snow and begins to interview the families of the ‘suicide girls’, some of whom, it seems, were associated with the ‘headscarf girls’ – a loose band of young Islamists who have come into conflict with the authorities over the government’s ban on headscarves being worn in secular institutions. Ka finds the ‘headscarf girls’ difficult to spot, since he has ‘not yet acquired the secular intellectual’s knack for detecting a political motive every time he saw a covered woman in the street’. But, even when he sees the town’s education director being murdered for enforcing the headscarf ban, Ka doesn’t seem especially worried that he might be getting out of his depth: he’s not all that interested in his journalistic mission, and his real purpose in Kars is to woo Ipek, the hotel owner’s daughter – an old acquaintance from his left-wing student days.

Snow continues to fall until Kars is entirely cut off from the outside world. Ka walks the frozen streets. Contact with Ipek, who seems to reciprocate his interest (‘I, too, want to make love, but ...’), lifts his spirits: for the first time in years he begins to write new poems. Enduring frequent power cuts and drinking endless cups of tea, he meets the town’s notable characters. First comes Muhtar, Ipek’s ex-husband, another old acquaintance; a former leftist, he’s now running in the local election as a moderate Islamist. Muhtar, a failed poet, speaks about the cultural shortcomings of the sheikh who converted him: ‘He knew nothing of Modernist poetry, René Char, the broken sentence.’ His disappointment in his sheikh led him to join the less fiery Prosperity Party, for which ‘my Marxist years prepared me well.’ Ka also runs into Necip and Fazil, a pair of students from the religious high school, both of whom are in love with Ipek’s sister Kadife, the leader of the ‘headscarf girls’. (Necip also wants to be the ‘world’s first Islamist science-fiction writer’.) Then, in utmost secrecy, Ka is summoned to a meeting with Blue – ‘a political Islamist of some notoriety’. Blue is on the run, accused of arranging the murder of an ‘exhibitionist’ quiz show host who ‘uttered an inappropriate remark about the Prophet’ on live TV. He is also darkly handsome and, despite his anti-Western views, contentedly addicted to Marlboro Reds.

There’s something theatrical about these discontented people. Blue, in particular, revels in his own mystique. When he first went on TV, ‘he was such a hit as the “wild-eyed, scimitar-wielding Islamist” that he was invited to repeat his performance on other channels.’ Leading Ka to Blue’s hideout, Necip strikes ‘a pose straight out of a comic-book’; Kadife, ‘a pose from a second-rate Turkish film’. But the forces of muscular secularism, headed up by Sunay Zaim, are even more histrionic, making Ka feel like an actor in a ‘dated play’. Sunay, an ageing hero of 1970s theatre, was once admired for his stirring portrayals of people who ‘would and could bring happiness to the people through the exercise of merciless violence’ – Lenin, Robespierre and so on. His fortunes declined after the 1980 coup, when he over-exuberantly suggested that he might be the man to play both Atatürk and the Prophet. He’s spent the last few years playing low-rent dives, perfecting his own brand of ‘Brechtian and Bakhtinian’ theatre. This involves enlivening antiquated republican plays with topical skits and belly-dancing interludes.

Taking advantage of the closed roads and the absence of the city’s senior military officers, Sunay and his troupe put on an evening of entertainment. Local bigwigs and boys from the religious high school attend. So does Ka, who reads a poem, but the main attraction is a Kemalist period-piece called My Fatherland or My Headscarf, which doesn’t impress the religious part of the crowd. Heroic republican troops storm the stage to rescue the bareheaded heroine from the play’s ‘bearded, prayer-bead-clutching’ villains. The schoolboys in the audience start to jeer, and the troops open fire on them with live rounds. Necip is killed, as are several others. Sunay takes control of the city, and a ‘special operations team’, led by the sinister Z Demirkol, starts rounding up Kurds and Islamists. Ka finds himself acting as an intermediary between the city’s political factions – chiefly, it seems, in order to persuade Ipek’s Communist father to go to a meeting, thereby giving Ka a chance to lure Ipek into bed. But Ka’s low aims and misplaced self-confidence soon lead him into trouble. There are aspects of Ipek’s romantic history he hasn’t yet found out about, and though he likes to imagine himself as ‘the sad romantic hero of a Turgenev novel’, one of the book’s epigraphs, more ominously, comes from Conrad’s Under Western Eyes.

Snow also has an epigraph from Dostoevsky’s notebooks: ‘Well, then, eliminate the people, curtail them, force them to be silent. Because the European Enlightenment is more important than people.’ And it’s not hard to see the attraction, for Pamuk, of the Russian novelists as models: if the paired protagonists of The White Castle make unlikely avatars of Proust’s narrator and Albertine, there’s no such problem when the characters in Snow start pursuing the local version of the Slavophile v. Westerniser debate. ‘Ka loved Turgenev and his elegant novels; and, just like the Russian writer, Ka had tired of his own country’s eternal troubles and had come to despise its backwardness, only to find himself gazing back with love and longing after he’d left for Europe.’ But, caught between the likes of Blue and Z Demirkol, his gentleness finds its limits. Blue, the terrorist, who ‘will never manage to do anything in life but generate more wrath’ – and who contends, in his own way, that two plus two equals five – is allowed to score points off him: ‘Perhaps from time to time you say a word or two reproaching the tyrannies visited on the Islamists and the Kurds, but in your heart of hearts you don’t mind at all when the military takes charge.’ In Ka’s case, at least, Blue turns out to be right, and by the end of the story the enlightened cosmopolitan has made a lethal bargain with the powers that be.

Pamuk does more than unwrap his protagonist’s bad faith, however. He is sardonic about the way the Islamists took over from the Turkish left as the party of the country’s many have-nots. He also has some uncomfortable fun at the expense of the education director’s pious assassin, who tells his victim that ‘the celebrated film star Elizabeth Taylor ... might have known some happiness’ if she’d taken the veil. (‘Why are you laughing, sir?’) Kars is haunted by its Armenian history, and Pamuk has Ipek discuss a local museum commemorating ‘the Armenian Massacre. Naturally, she said, some tourists came expecting to learn of a Turkish massacre of Armenians, so it was always a jolt for them to discover that in this museum the story was the other way around.’ The first half of the novel is enjoyably Gogol-like: conversations are interrupted by blackouts, like the mist that obscures the first section of ‘The Nose’; low-level secret policemen complain about their comically futile missions, and no one fails to comment on Ka’s luxurious overcoat. As for his original mission, investigating the causes of the high female suicide rate: it turns out that the Islamists and the secular government are united in not wanting the issue discussed.

After Sunay’s ‘coup de théâtre’, the city’s underground factions get together to thrash out a statement of protest for the Western papers. Ipek’s father causes some confusion by asking: ‘If a big German newspaper gave each of you personally two lines of space, what would you say to the West?’ Only Blue has any idea what he’d say, but eventually a Kurdish youth pipes up: ‘We’re not stupid! We’re just poor! And we have a right to insist on this distinction.’ No one is very impressed by this but, later, when Orhan-the-narrator hits town, he runs into Fazil, Necip’s friend, who’s now working as a hotel receptionist. ‘I can tell from your face,’ Fazil says,

‘that you want to tell the people who read your novels how poor we are, and how different we are from them ... If you write a book set in Kars and put me in it, I’d like to tell your readers not to believe anything you say about me, anything you say about any of us. No one could understand us from so far away.’

‘But no one believes everything they read in a novel,’ I said.

‘Oh, yes, they do believe it,’ he cried. ‘If only to see themselves as wise and superior and humanistic, they need to think of us as sweet and funny, and convince themselves that they sympathise with the way we are and even love us. But if you would put in what I’ve just said, at least your readers will keep a little room for doubt in their minds.’

Orhan promises to put what he’s heard in his novel.

*Letters · 23 September 2004*

In his piece on Orhan Pamuk, Christopher Tayler says that I have ‘a wayward sense of register’, that my ‘English sentences are often hard to decode’, and that my ‘translations read very awkwardly in a way that all the other translations of Pamuk do not’ (LRB, 5 August). In 1998, Professor Talat Halman said of my translation of The New Life: ‘Güneli Gün … has done an impressively successful translation, faithful and idiomatic. Some critics have characterised the translation as stilted at times, but this is hardly Gün’s fault. Pamuk … occasionally writes some awkward sentences. If anything, the translation has managed to expurgate many of the careless clauses.’ Pamuk, who keeps a sharp eye on his critics, has been modifying and simplifying his language in his last two books, My Name Is Red and, more recently, Snow. But vintage Pamuk can be hard to read. Other translators of his work have phoned me, thanking me for unpacking some of his more enigmatic sentences. Yet I was unwilling to disrupt his prose, because every linguistic puzzle he presented seemed worth solving.

Güneli Gün | Oberlin, Ohio

## A detailed tapestry of 16th century Turkey

By Sarah Coleman Dec 9, 2001 SFGate

*My Name Is Red By Orhan Pamuk*

*Alfred A. Knopf; 417 Pages; $25.95*

Anyone who's visited Turkey has probably been drawn to the charm of Turkish miniature paintings. These delicate, stylized images of battles and bathhouses,

with their fine lines and flat colors, are exquisite examples of Eastern artistry. Looking at them is like peering into an exotic and radiant dollhouse.

In Orhan Pamuk's "My Name Is Red," a 16th century Turkish illuminated manuscript is at the center of a historical murder mystery. Pamuk, a best- selling author in Turkey, uses the history of his country's art to examine intersections between religion, creativity and human desire.

The result is a huge and ambitious novel that is by turns charming and pedantic. Like Umberto Eco's "The Name of the Rose," "My Name Is Red" combines down-and-dirty intrigue with scholarship and a postmodern sensibility. Written from multiple perspectives, it includes chapters narrated by recently murdered people, a dog, a tree and even the color red.

"I am nothing but a corpse now, a body at the bottom of a well," begins the compelling first chapter. The body belongs to Elegant Effendi, one of four master artists who has been commissioned, at the end of the 16th century, to illustrate a secret and controversial manuscript for Ottoman Sultan Murat III.

Unlike other Turkish illustrations, this one will incorporate the newest techniques from "Frankish," or Venetian painting, using perspective, shadows and -- most daring of all -- recognizable portraits of individuals.

To show how such artwork might threaten the social order, Pamuk sketches a lively Istanbul at the end of the 16th century. It is a time of plagues, fires and war, where religious repression coexists with decadent social and sexual behavior. Fundamentalism is gaining ground: A preacher called Nusret Hoja is making political capital by attributing Istanbul's corruption "to disregard for the strictures of the Glorious Koran, to the tolerance toward Christians, to the open sale of wine and to the playing of musical instruments in dervish houses."

According to Nusret's dogma, the individual expression in the new manuscript amounts to blasphemy against Allah. Its commission has thrown the community of illustrators into a panic, sowing division between the Islamists (who include the murdered Elegant) and the free-thinkers.

All of this allows Pamuk to explore the aesthetics of representation in great and sometimes exhausting detail. We learn about the significance of gilded borders, prescribed ways for drawing eyes and nostrils and the tension between innovation and imitation. "If the picture is to be perfect," says one illustrator, "it ought to have been drawn at least a thousand times before I attempt it." Others in the novel refuse to embrace such dictates. Enishte ("Uncle"), who is coordinating the secret manuscript, thinks that Western portraiture is the way of the future, though he acknowledges the danger of an art that glorifies individual humans.

Enishte charges his nephew Black with uncovering the identity of Elegant's murderer, knowing that Black will be motivated by his love for Enishte's daughter Shekure, a beautiful mother of two whose husband is missing in action in the Persian wars.

When Enishte is murdered, the stakes are raised, and the narrative becomes taut. The second murder scene is a wonderful set piece in which the color crimson dominates, as it overflows from the murder implement (a bottle of red ink) and in the blood that gushes from the victim's head.

In Shekure and Black, Pamuk has created compelling characters whose voices jump off the page. He also introduces another colorful narrator in the person of Esther, a Jewish fabric peddler who acts as an intermediary between the two lovers.

Unfortunately, the chapters related by these three only make up around a third of the book. Others, narrated by inanimate objects such as the drawn figures of horses and dogs, often lapse into dry art history seminars.

Perhaps Pamuk's greatest misstep, though, is that he fails to adequately characterize the three main murder suspects. These are Elegant's fellow illustrators, and although they are distinguished by the colorful nicknames of Stork, Butterfly and Olive, it's hard -- up to the final chapters -- to tell them apart. Instead of giving us scenes from their lives, the three men offer convoluted riddles about illustration. In his urge to make all of them viable suspects, Pamuk blurs the lines among them, thereby lessening the reader's involvement in the resolution of the murder case.

This is a shame, because at its best, "My Name Is Red" contains chapters of stunning artistry and drama, and it offers a fascinating view of life in a historic Istanbulite artist community. Like Calvino, Borges, Kafka and Eco (to all of whom he's been previously compared) Pamuk is a writer who is able to combine avant-garde literary techniques with stories that capture the popular imagination.

Here, the ingredients are potent, but the balance is off. Like an overenthusiastic master illustrator, Pamuk paints a vivid picture, but loads it with so many details and symbols that the eye has nowhere calm to rest.

## Building a multi-tenant SaaS data model

Tim Nolet Chief Evangelist,

**Chekly**: Code-first synthetic monitoring for modern DevOps. Monitor your APIs and apps at a fraction of the price of legacy providers. Powered by Monitoring as Code and Playwright.

March 17, 2019 (Updated: February 9, 2024)

https://www.checklyhq.com/blog/building-a-multi-tenant-saas-data-model/

This is a run down on the basic multi-tenant SaaS data model underlying Checkly. Users, accounts, plans, that type of stuff. When building this, I found it surprisingly hard to find any solid info in the gazillions of developer and startup blogs; most were just to vague on the implementation details.

**PostgreSQL, Mongo or DynamoDB?**

Short version: Postgres. It's an absolutely fantastic piece of engineering.

Run it on Heroku, it just works™ and it makes me feel like some grizzled, elite Spetsnatz commando has my back.

Long version: In an alternative universe, I'd have implemented Checkly completely on AWS DynamoDB. The peace of mind a fully managed — web scale... — database brings you is just awesome when you want to focus on customers instead of database management.

So I actually tried getting a SaaS model into DynamoDB. That exercise was so utterly unsuccessful it actually makes me laugh now. The limitations on the primary key and range keys alone make it just a very bad fit for this use case. AWS has a write up on how something like this could actually be achieved. I have never seen a better example of "shoehorning".

MongoDB then?

This could actually have worked. Still no actual relations, but Mongo has ways of making this work. Sadly, my brain just can not get used to the MongoDB query language: I found the defacto Node.js driver Mongoose a convoluted mess, especially the documentation. Also, I knew I was going to do a ton of aggregation on metrics (averages, percentiles etc.) and Mongo’s aggregation features were/are very minimal.

DynamoDB and also MongoDB are completely valid products. Just don’t use it for anything relational(ish). Mind blown.

Also, Postgres is pretty nimble, even with terabytes of data and high read/write traffic.

So Postgres it was. Onwards!

**Single, multi-tenant database**

Short interlude before we dive into the actual data model. How do we handle the partitioning of customers at the very highest implementation level now that we have chosen Postgres?

There are a couple of options out there:

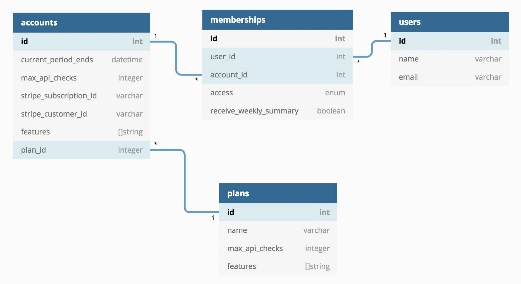
**One tenant, one database**: Each tenant its own database instance. 100% separation. If you expect either very high security requirements from customers or expect the need to scale out parts of your platform for specific customers, think Salesforce like size.

**One database, schema per customer**: A single database, with a separate schema for each customer. Duplicating tables for each schema. This can make migrating especially heavy users to higher spec machines easy. It also is even more secure as you leverage the built-in security model of your DBMS.

**One database, one schema**: Just one schema that holds all the tables and an “account ID” for each tenant. This is the model we chose for Checkly, as all of the above concerns are/were not applicable to Checkly. The expected usage patterns and storage needs are also not that massive

**Accounts & users**

Checkly and many other typical B2B focused SaaS are structured as follows:

1. Users belong to an account, also sometimes called an organization.
2. An account can have multiple users and users can switch between multiple accounts. This is specifically interesting if offering a product aimed at agencies or larger organizations where one customer can actually "own" multiple accounts.
3. Accounts have settings, preferences and resources associated with it. They apply to each user linked to that account.
4. Users can set personal preferences per linked account.
5. Users can have specific access rights per account.

To achieve this, we use the following data schema:

Let's go over each table and point out what's interesting:

**1. Users**

The users table is quite straightforward. It holds the basic details we gather either from social login (Google & Github in our case) or from email signup. Notice there is no password field. We use Auth0 for that.

**2. Memberships**

The memberships table models the many-to-many relation between users and their accounts. The most important bits are:

* Access control: The access field is an enum that holds one of the possible user roles as a string constant, i.e: READ\_ONLY, READ\_WRITE, ADMIN or OWNER. The actual enforcement of these roles is done in the API but that's a story for another time.
* Per-account user preferences: The receive\_weekly\_summary field is an example of a specific setting the user might want to toggle per account.

**3. Accounts**

This is where the meat is. The primary key of the accounts table is referenced as a foreign key in many, many other tables. Invoices, dashboards, team invites, etc. all reference the accounts table.

Besides being the root for almost all resources in the whole application, the accounts table has three main functions:

**Subscription & billing cycles**: we need to keep track if payments are made in some way. In the example you can see the stripe\_customer\_id, stripe\_subscription\_id and the current\_period\_ends fields.

When a user signs up for a paid plan, we use Stripe webhooks to fill these fields. On each subsequent billing cycle our API receives another webhook call and we bump the current\_period\_ends date field.

Notice that if for any reason no money comes in, the date is never bumped. The rest of our code uses this field to check whether it should allow users to keep interacting with the product and whether it should do background jobs.

**Plan limits**: Each plan customers can subscribe to has volume limits associated with them. In our case these are things like amount of checks, amount of team mates, SMS messages sent per month. The upper limits for this are recorded in field like max\_api\_checks from the example.

These limits get set when subscribing and when changing to a higher/lower plan. More on that below.

**Plan feature toggles**: Plans can also have non-volume based differences. For example, people on our Developer plan do not have access to Setup & Teardown scripts. These are essentially feature toggles based on what plan you subscribe to.

To model this, we use the features field which is of the String Array type, a pretty nice feature in Postgres. It holds a set of string constants like SMS\_ALERTS, SETUP\_TEARDOWN and PROMETHEUS. Each constant is associated with a feature and we use them to enforce the limits in the API on the backend and show a nice hint to the user that a feature is only available after upgrading.

Funky tip 👉 We actually bump the current\_period\_ends date + some extra grace days. Would suck if the monitoring is down due to annoying but benign some card issue.

**On plans & subscriptions**

This might seem unintuitive, but there is a very weak relation between the plans table and the accounts table — you might have noticed the duplication of the max\_api\_checks and the features fields in each table.

Both these fields describe plan limits and we actively copy values from the plans table to the accounts table when an account is created. This makes tweaking specific customer requests a lot easier.

Want a custom plan that has slightly more API checks? We can do that. Just update the specific account row and adjust the monthly price in Stripe. Want to give access to a new feature to a small set of users? Just update the features array for just those account. Later roll it out to all customers. Who needs a complicated feature toggling framework?

So what actually is a subscription in this context? Good question, as it has no direct representation in this model. In the most literal sense, it is the invoice.payment\_succeeded Stripe webhook coming in and telling us customer ID X with subscription ID Y has just paid.

Everything else is decoupled: the actual price per month / per year, the billing interval, what features you get and what volume limits apply. This decoupling then allows you to experiment more with price and the exact composition of your plans. Actually, the whole Checkly application has no "knowledge" of the pricing except for some markup on the https://checklyhq.com/pricing page.

## WebAssembly:

A promising technology that is quietly being enshitified (kerkour.com) <<blocked; p’haps intentionally>>

53 points by romaintailhurat 1 day ago | 58 comments

|||coxley||| On a version of Chrome from two days ago, and still blocked from visiting the site. What an aggressive, false positive, worthless check.

|||worik||| OT but this...

> I strongly believe that async is the new billion dollar mistake of the 2020's: a design aberration that wasted so much developers time that it had cost billions of dollars to companies.

Yes. Nicely put

|||mbStavola||| Completely disagree.

Async as a paradigm has probably created billions of dollars in that it's allowed many many many developers to reasonably scale beyond what they could've otherwise.

|||anakaine||| Agree wholesale. I've got a complex data reader built on a single server that receives tens of thousands of requests a day, and most tend to happen in a few hour window. Async was super important to scale to user requests by allowing non-blocking calls to not get captured behind blocking ones.

|||tommiegannert||| I think GP is arguing that green threads e.g. in Go is a better way to structure the same thing.

|||worik||| No. I had a nasty experience with Typescript's async/await and I am annoyed

I find async/await very confusing. It as one thing pretending to be another

I prefer my paradigms simple and straight

Async await is deceptively simple but very wiggly. Not straight at all. Loads of magic fairy dust obscuring what's really happening

It is helpful until it isn't and then it's very unhelpful

|||prerok||| And I disagree with your disagreement :)

Just some background first: one of the billion dollar mistakes is the null pointer which is a special value to be assigned to any type. It was inherent in the first high (today considered low) level languages. Those languages (C most importantly, then Java and nowadays even Go) enabled many more programmers to create programs and they likely would not have been able to, had they been forced to write assembly. So, it's a billion dollar mistake within a trillion dollar revenue.

The async now solves a similar practical problem but in the space of prallelism (or concurrency, in some cases). A different approach might have given us the same result, but a different implementation that might be just as approachable.

Optionals and Results have given us a better way than to have nulls, and maybe golang style channels could have given us a better way to handle async?

|||amw-zero||| What does this even mean, how can you do any form of modern computation without async?

|||MrDarcy||| async / await as they exist in JavaScript, Python, and Rust aren’t the only way to execute tasks concurrently.

|||eureka-belief||| But they are a darn convenient mental model if you’re just trying to do some basic async. The alternatives are harder to wrap your head around. Unless I’m missing some other cool pattern.

|||amw-zero||| That’s what I’m asking - did the comment mean that the async/await pattern is bad, or that asynchronous programming in general is bad?

|||aixpert||| Wait they promised to not adopt the cancerous function coloring approach in wasm. Please tell me they didn't follow the dark path of JS

|||shanemhansen||| > You have been blocked! If you are not a malicious actor, please update your web browser to the latest version to access this website. If the problem persists, please contact support.

Unfortunate.

I think that these days the scammers have won. They can do a better job than I can at being authentic.

FWIW based on the comments I actually prefer a richer runtime that allows some of the language level features that have been hard to get support for. I think having mainstream support for effects would be sort of magical.

Actual isolated modules? Talk about living in the future.

Wasm is a lot like the jvm and flash, which makes some people hate it. But for me the problem with those platforms was the execution, not the idea.

I'm cautiously optimistic that wasm will finally give us a good high performance cross platform secure execution target for the web.

|||shishcat||| You have been blocked! If you are not a malicious actor, please update your web browser to the latest version to access this website. If the problem persists, please contact support.

|||Arnavion||| Components are basically COM / CORBA for WASM. Just like COM enabled multiple languages to exchange structs and function calls with each other, WASM components do the same.

The C ABI approach works fine but becomes complicated when you want to pass things that aren't fixed-length, because unlike the C ABI usage when linking different libraries into the same address space, the host and the module run in different address spaces. Eg if the host wants to pass a string to a module fn, it first needs to call a different "malloc" module fn to allocate some space in the module address space and return that address, then the host writes the string to that address, then calls the original fn it wanted to call with that address, then calls a third "free" function at the end to release that allocation. Does that work? Of course it does. But the component ABI abstracts over all that for you and lets you call the function with the host's string type, and the runtime on the host side and bindings on the module side handle all the thunking so that it appears on the module side as the module's string type.

TFA doesn't seem to have any arguments other than "It's too many new words for me to learn so it's unnecessary I decided." Eg:

>And don't get me started on WIT's kebab-case identifiers (function names, interface names...). Why??? How can a specification about cross-language interoperability can come with a convention that basically no programming language use?

The point of the kebab case is that it does \*not\* show up in the generated bindings. It's processed by the bindings generator into the appropriate casing for the language. Eg a function name `foo-bar` becomes `foo\_bar` in Rust, and a type named `foo-bar` becomes `FooBar`.

|||wrs||| I think the argument here is that you don’t need a full N:1:M mapping of data types to make this work. Transparency is a myth. If you try it (as in Sun RPC, DCE, CORBA, COM, etc.) you end up just writing wrappers anyway as you hit some edge case that the universal type system doesn’t support. E.g., look at the clunky IDL interfaces COM had to invent to get interop with JavaScript objects because IDL had baked-in assumptions about how records work.

Since the point is to enable component-level interop, not fine-grained function interop, you can instead define a simple interchange format (more JSONish than IDLish) and write the library interfaces accordingly. More of a microservice approach than a linker approach.

|||Arnavion||| Counter-point: I happen to be working at $dayjob on things related to Components (both on the host side and the module side), including using some "WIP" things like the "resource" feature (opaque handles with methods), and haven't hit any edge cases.

|||wrs||| Sure, it’s expected that you won’t hit problems if you’re using the paradigms that existed when the IDL was defined. On the other hand, trying to apply Sun RPC after everything became object-oriented…well, that’s when we blew it up and invented CORBA. So I’m just saying the cycle is likely to continue with this approach. In other words, come back in ten years and see how it aged.

The alternative is to use something so simple it can’t become outdated. Dumb down the interface rather than making the interop omniscient. (Kind of like the dumb network principle.)

|||wrs||| I just realized: The very fact that WASI isn’t using any of the previous N attempts at a supposedly universal IDL, but instead inventing yet another one, itself demonstrates the problem with this approach.

|||torginus||| It's entirely reasonable for WASM to use its own IDL. Afaik Chrome already uses an IDL to describe the interaction between native C++ classes and their Javascript bindings.

Integrating WASM modules with each other, WASM with JS, and WASM with native functionality seamlessly is a huge and unsolved challenge.

|||p\_l||| Or it just points towards NIH syndrome, combined with hype and anti-hype driven mentality of programmers, and general forgetting of previous technologies. Or even when the differences are close to 0, doing a new not exactly same implementation for ... whatever reason.

|||pjmlp||| Or COM's evolution, WinRT, where the .NET metadata also doesn't really map to JavaScript or C++, without some additional kludges with metadata as workaround.

|||p\_l||| Don't forget the extra fun that happens when you call free() from a different allocator than the one that allocated the memory object, though I suspect WASM's module approach might make it difficult to happen...

|||Arnavion||| Yes, that won't happen here, because the host is just calling some global malloc and free exported by the module. Well, obviously the implementations of those globals can be buggy in some way, but that's the module's bad.

|||pjmlp||| You have been blocked, apparently Android Chrome is a malicious browser.

|||loco5niner||| Disabling Ublock Origin for that site worked for me.

|||fathyb||| Just had it too, refreshing worked.

|||pjmlp||| Not on my case.

|||torginus||| Probably they are blocking access from 'suspicious places' e.g. anywhere that's not the US or Western Europe.

The irony of talking about enshittification...

|||pjmlp||| Germany...

|||smolder||| I wish people would stop using the term "enshittify" to mean other things than what Doctorow coined it to mean. This has nothing to do with "enshittification" and there were plenty of other words to pick from.

I see the article says "sabotaged", so perhaps that was fixed? Or the titles never matched?

|||syrusakbary||| It's refreshing to see that we are not alone in our thoughts on how the community is not being stewarded towards its own interests. I applaud the author on how clear he made the argument.

For those that aim to continue working on top of WASIp1, WASIX (https://wasix.org) might be a great way to get your programs with sockets and threads fully running on Wasm.

Note: I work at Wasmer (https://wasmer.io), a WebAssembly runtime.

|||aixpert||| The WIT types are a great proposal: we need wasm to be able to pass around strings and lists! Maybe should have been part of the MVP to make wasm a success.

This whole component model mess on the other hand though: Don't get me started with all these kebab case worlds

|||superchris||| The promise of being able to use any library from any language is really quite compelling, and that's what the WASM component model is about for me. It's pretty sad to say this amount of hate TBH

|||pjmlp||| Already fulfilled by other bytecode runtimes in the past since UNCOL (1958), eventually people settled on a couple of key languages instead.

Many of us (haters), are old timers seating on the saloon bench seeing yet another slew of gold diggers arrive full of enthusiasm into town.

This time is going to be different, and it will take over the world (TM).

|||jauntywundrkind||| > The promise of being able to use any library from any language is really quite compelling, and that's what the WASM component model is about for me.

I'm excited for having extremely lightweight sandboxed. WASI enables having a runtime with a bunch of loaded libraries. You can start a very small script & link in the already loaded modules on the fly, in a very secure fashion.

It's be like having an isolate-oer-requesf model. Super secure, but with fantastically low overhead. Ideally instead of having a huge app server, a front end router would be picking which specific actions to run.

The ability to spin up a cast number of very lightweight secure processes and have them communicating with each other is fascinating. App servers as we build them are ghastly complicated swiss army knives, and being able to have something like a "serverless"/lambda architecture where we can narrow the scope down & really think about what has to be in a given request handling's process could be a big operational boon, if we're willing to once more venture away from the comforting warmth of the monolith that folks love huddling up next to.

As usual though hope & possibility is speculative & nuanced & diverse, and disbelief & disgust is blanket & unifying. I have no clue how I'm still so shocked to see negativity upvoted, positivity out down upon, after it happening so many times but I keep being surprised how strongly negativity reigns. And how fiercely & widely it downvotes! There's just something about the disbelievers & skeptics that they have to smash the downvote, can't abide possibility or excitement; there's never any wait and see, never any maybe about it. Just doom & gloom on and on.

|||kmeisthax||| To be clear, this is (mainly) about the enshittification of WASI, not WASM. If you're writing code to run in a web browser you will never interact with WASI. It is unfortunate to see WASI fall victim to the software componentry / IDL meme, but I doubt we're going to see that be an issue in browser WASM.

The part that actually pertains to WASM - async/await - exists because everything in a browser lives in Someone Else's Single Threaded Event Loop. Writing code that lives in an event loop is pure pain, and async/await exists solely to fix this problem. The billion dollar mistake is not adding async/await to programming languages, it's single-threaded event loops. Anyone talking about C10K or non-blocking I/O in regards to async/await is probably missing the point, because aside from one stubbornly single-threaded programming language[0] you can launch threads to handle reading or writing data and sockets.

[0] JavaScript. It's always JavaScript.

If you were thinking Python, you're wrong. Even with the GIL, Python supports threads, and concurrent I/O is one of the few reasons why they're useful.

|||pjmlp||| You definitely will, because this is the approach that is also being taken by runtimes like Dart, Kotlin among others.

|||worik||| > Writing code that lives in an event loop is pure pain,

Do you think?

I find explicit event loops pleasurable.

Does that make me a freak?

|||coxley||| They're pleasurable when you're the only author. They become a minefield unless every contributor is diligent, which happens very often.

|||curioussavage||| Yup. Somebody is going to write something that hogs cpu without yielding to the loop or queues up an absurd number of tasks waiting to be executed which effectively has a similar effect. All of a sudden latencies are high. Depending on how tracing is done it can appear like certain io operations are the culprit if you just go off of traces.

P99 latency for every route on your web server will be fixed at the max time any individual unit takes to execute before yielding plus loop overhead and its own time. This can drastically increase it for many routes.

Meanwhile some genius insists that our app is “io bound” so the single threaded async runtime must be a perfect fit.

Apart from just being generally faster at least with go I know that when somebody screws up there should be n other threads still executing tasks.

|||TazeTSchnitzel||| But WebAssembly is one of the most promising opportunities in a long time to escape the C ABI problem. Why should we squander that? Do we want to be stuck writing C forever?

|||adastra22||| Why not just come up with a new calling convention and implement it for a handful of popular languages?

|||pjmlp||| Some of us already escaped that, with other bytecode formats, or stuff like COM, AIDL, XPC, D-BUS.

|||fathyb||| > If you are using WebAssembly in a web browser then you are good, WASI does not concern you.

In case you absolutely do not care for WASI but love WebAssembly for the web.

|||olsonjeffery||| I don't think there's any guarantee that browser vendors will continue to support WASI 0.1 indefinitely; Once multiple versions of the standard are in play, codebase supporting them simultaneously explode in size. None of this gets into the likelihood that, once WASI hits 1.0, the prior versions very well could be "retired".

And keep in mind that the perspective of the author is that things will only get worse from hereon, with regard to WASM.

|||fathyb||| I didn't know browser vendors supported WASI! I said that as some of us do not use WASI at all, just WebAssembly itself, so they can save some time by not reading this if they're not interested in WASI and/or its limitations.

|||Arnavion||| Browsers don't have any built-in support for WASI. It's up to the web page to define those as exports for WASI-expecting WASM modules to import.

|||olsonjeffery||| It's hard because, a lot of this stuff (async WASM, effect-driven WASM, GC in WASM, etc) is all driven by the desire to push these concerns out of wasm bytecodes, because it explodes the binary's size.

I have a pretty trivial webapp written in Yew and I stopped working on it once I saw that the wasm artifacts were weighing at 4MB (uncompressed) for relatively little functionality. THIS is what is driving the next round of "work" on WASM: to wring more functionality out of a system (WASI 0.1) designed as a drop-in replacement for emscripten output. As an aside on this point, the author is waaaay in front of their skies with basically all of their critiques around ByteCode Alliance; A lot of innuendo to basically serve a rhetorical point that isn't really true (WASI 0.1 is "good enough"). It has drawbacks! That's what the iterations on the protocol are exploring!

And this tech is cool and all, but right now a reasonably-vanilla typescript react app w/ a modern bundler (ie including stuff like router, redux, oidc, etc) is beating the breaks off of a similar app in Rust or C#/Blazor in terms of bin size. And there's no perf or API-surface argument that overcomes this. And it has chilling effect on developers when they reach for this tech.

|||10000truths||| Then maybe "this stuff" should not have been tacked onto the WASM standard in the first place? There's no free lunch. Sure, it sucks that your favorite garbage-collected interpreter/runtime bloats your module size. But the alternative option is to bloat every WASM engine by shoehorning your use case into the standard.

|||fathyb||| It really depends on what you're using. If you use Rust with `wasm32-unknown-unknown`, you'll likely get small binaries (<200 kB). If you use C++ and Emscripten with all features enabled, then yeah you'll have multiple megabytes with all the libcxx and musl stuff.

|||olsonjeffery||| My direct experience with wasm32-unknown-uknown contradicts your own. I am building a yew on nightly, moving between release or debug affects nothing.

|||fathyb||| > moving between release or debug affects nothing

I'd start with that, it's an obvious red flag. Switching between both should create huge differences. Also look into `wasm-opt` from the Binaryen project for post-link optimizations, `wasm-ld` from LLVM isn't that great at DCE.

|||olsonjeffery||| Thanks for the feedback.

I was wrong about debug vs release; It's the difference between 4.0MB & 893KB on a ~500 LOC rust codebase (per cloc).

I want to observe that your suggestions don't undermine or refute my point about problems that WASM adoption faces. This serves to underline that the developer experience needs work. Opinions may differ on how much. Also that "developer experience" runs on multiple axes of concern (bin size, perf, accessibility, utility of language, etc).

|||fathyb||| I agree, my personal take: if you don't want to get your hands dirty, WebAssembly is not ready for you yet. It'll take at least 5 more years before the tooling gets into a state where things should just work (especially DWARF support). I mean, we still cannot free memory! (actually there is a crazy way by recreating a new WebAssembly instance with a shrunk'd `ArrayBuffer`, but it requires you writing your own memory allocator)

My point is: if you're comfortable working with a slightly obscure microcontroller, then you won't have much problems. LLVM supports WebAssembly out of the box, so it mostly feels like programming for one of those.

Anecdotally: we run a large Rust app in under 1 MB of WebAssembly at Zscaler.

|||worik||| > 4.0MB & 893KB on a ~500 LOC rust codebase (per cloc).

My guess is 50,000 LOC would not be 100 times bigger err

Methinks you are counting overhead

More accurately... meguess

|||tyzoid||| > We're sorry but this website doesn't work properly without JavaScript enabled. Please enable it to continue.

|||jauntywundrkind||| > It means that the scope of WASI has shifted from falling language B from language A to solving something that nobody has asked for.

Uhh really? You just got done pointing out

> But, WASI 0.1 was severely limited about the type of data that it could exchange: basically integers and pointers to buffers.

We obviously couldn't stop there.

The author has all sorts of shit to throw about the interface definitions being way more complex than they want.

It turns out calling other languages isn't just about calling their stuff. You also need to be able to import and then latter link the things you want to use. Being able to say what it is you want brought in, what you want to link to, that's stuff the runtime needs to be able to do.

Look at c and c++. They've been around for decades and there's still nearly no mainstream package/library management. Because it looks like what Kerkour asked for, because everything is too simple in that world.

You need this stuff. You need to be able to create high level semantics to interoperate across. You need a rich enough ABI to let languages negotiate for & get the things they're going to call.

The qualms against async are even less well defined & even more unsupported, which is compensated for being ever more foaming at the mouth & wild gesticulating. Rather than acknowledge that yeah, some people like & use async in languages, there's utterly untargeted shade of the broadest degree:

> Most importantly, after all these years, nobody knows when to use [rust's async] or not! Should this library be async or not?

Io-less libraries that can put off being async are in fact excellent. But there still, in most systems, are things happening over time, and async happening in Python and Node and others radically upped the game of what was possible. And in the past decade have even evolved into something pretty nice & great & usable.

But Rust having some difficulty figuring out how to manage their ecosystem is, to Kerkour, apparently enough to damn the whole enterprise, "The new billion dollar mistake." Having been around for cgi-bin and mod-perl, I don't think I'm so massively massively confidently assured.

Things aren't always exactly as perfect as we might want. And that just really drives some people wildly mad, is inexcusable. 'We should never have tried, ruin to those trying to improve things, & drop it all & go back!,' seems to be the message. Actually the message here is even worse, more, 'these people are vultures trying to syphon money with the express intent to create waste & milk the anarchy,' which is some contagious shit to be spewing.

This seems so overblown. WIT's are not that complex. Theres reasons stuff is like this (permitting wasm engines form a registry to give people what they're asking to use, without needing internals that would have to DIY this all themselves). And maybe such a broad objective as a universal computing runtime might be possible with less. But honestly the risk of underbidding & failing to connect different systems well seems far worse than the risk of, I dunno, what the author seems to be apoplectic about, perhaps maybe enjoying using a package manager to help satisfy dependencies.

One thing the author is right about, that makes me extremely sad:

> I hardly see browser vendors implementing this

I can excuse this today, because it is unknown. But wow it's scary as hell seeing how uninvolved & interest the browsers look. The browsers seem full speed ahead building their own take on platform, doing their own file system APIs and what not. There's great efforts by many to bridge the two worlds, to create WASI runtimes for the browser, but some day I really hope the promised universal machine of WASI is something the web can enjoy and use. But I'm also not in a hurry; I think there's a lot of figuring how best to make this wasi 0.2 world nice & letting languages figure out their own tooling to come play. Patience & improving & iterating is good; trying is good. We improve & progress through time and effort.

## Premature Optimization (c2.com)

Premature optimization is the root of all evil -- DonaldKnuth

In DonaldKnuth's paper "StructuredProgrammingWithGoToStatements", he wrote: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

However, PrematureOptimization can be defined (in less loaded terms) as optimizing before we know that we need to.

Optimizing up front is often regarded as breaking YouArentGonnaNeedIt (YAGNI). But by the time we decide that we need to optimize, we might be too close to UniformlySlowCode to OptimizeLater. We can use PrematureOptimization as a RiskMitigation strategy to push back the point of UniformlySlowCode, and lower our exposure to the risk of UniformlySlowCode preventing us from reaching our performance target with OptimizeLater.

For those who don't work to strict memory or CPU cycle limits, PrematureOptimization is an AntiPattern, since there is only cost and no benefit. For those who do, it is often confused with poor coding, or with misguided attempts at writing optimal code.

A common misconception is that optimized code is necessarily more complicated, and that therefore optimization always represents a trade-off. However, in practice, better factored code often runs faster and uses less memory as well. In this regard, optimization is closely related to refactoring, since in both cases we are paying into the code so that we may draw back out again later if we need to. We don't (yet) have PrematureRefactoring regarded as CategoryEvil.

Another common misconception is that any level of execution speed, or resource usage, can be achieved once the code is complete. There are both practical and physical limits given any target platform. PrematureOptimization is not a solution to this, but it can help us DesignForPerformance. When working in an environment where resources are less limited, this is unlikely to be a problem.

The opposite of LazyOptimization. -- DavidMitchell

*ThreadMode discussion:*

Optimization practices allow you to produce a better product. Customers like that. -- GeraldLindsly

Premature 'Operations Research' is almost impossible to do wrong. -- gl

It happens once in a while though... Consider Hiroshima and Nagasaki. -- UN

Holy Mackeral, there, Saphire! A new Wiki record! GodwinsLaw invoked on the third sentence! [Well, Nazis aren't mentioned specifically, but bringing US nukage into the discussion has the same effect, eh?]

Self-fulfill much? Your mention of GodwinsLaw was the first reference to Hitler or Nazis; Hiroshima and Nagasaki have nothing to do with them and no, it doesn't have "the same effect" other than, again, your attempt at self-fulfillment. Mention of Hiroshima and (especially) Nagasaki is legitimate and represents a whole class of "premature operations research" with disastrous consequences. P.S. If you would like to make a slightly less bad impression, learn to spell.

Sure, a little extreme, but isn't that a clear example of "PrematureSomethingOrOther" Apologies for jumping on CarHoare's headline there. Thought it would get some attention though. Thx. -- gl

I largely agree with ["premature optimization is the root of all evil"], but I have to wonder about some things... how would you feel about:

"Let's just make the app run as a simple single-threaded CGI executable. We'll get the whole thing finished and then it will be easy to refactor it into a J2EE server later on."

Some decisions have to be made early, or not at all. -- UnknownAuthor

Belated pessimization is the leaf of no good. -- LenLattanzi

I actually had a huge problem with premature optimization until I saw the quote above, and realized it was true. Computer games have always traditionally been heavily optimized by hand, and I used to optimize the most trivial pieces of code. I still shudder at string classes that copy the string data around.

It's a bad idea because of course refactoring and maintaining optimized code is very difficult. Pointer caching (e.g. using \*p++ in a for() loop instead of p[i]) makes the code less obvious, and changing the iteration becomes error-prone. Removing redundancy (e.g. if p == q + 50 is always true, eliminate q and replace with p - 50) suffers from similar problems, since redundancy in the first place exists to match the programmer's model.

One might say that code optimization is a form of refactoring in its own right, but which moves you away from what the language wants you to write and towards what the machine wants you to write. -- EddieEdwards

The pointer examples is especially interesting, because a compiler can perform them automatically over short regions of code! Furthermore, manual pointer management is much harder for compilers to understand.... If you have to optimize, try to avoid things that can be automated! -- UnknownAuthor

Also note that in C++ STL, using \*p and ++p is the recommended approach. Also, in a C/C++ for() loop you would rarely, if ever, have need for \*p++; you would place ++p in the for() loop "arguments" list [I don't know the correct name for the stuff that goes in the parentheses] and the \*p access would be in the body. -- WayneMack

Actually, this is not that unique to game programming. Anyone who has ever had to develop a product that meets the capabilities and performances shown in a RAD prototype has experienced this problem. You can do almost anything with demo software, but making it actually work is different. -- WayneMack

A former coworker of mine (who will remain nameless) once said, "Performance anxiety leads to premature optimization." How true. -- ApoorvaMuralidhara

The problem with premature optimization is that you never know in advance where the bottlenecks will be. Also, you make the design and the code very hard to modify later, when requirements change. Yes, requirements DO change. That's called maintenance and even for systems that are still in analysis, design and construction, that holds.

You can optimize a design in order to find the most fast way to solve a problem, only to find out later that your optimization doesn't allow the system to do what the customer wants it to do.

This can also be expressed as:

1. Make it work.
2. Make it right.
3. Make it fast.

This was expressed by one of the original XPers, AFAIK. (Yes; see MakeItWorkMakeItRightMakeItFast -- ed.) The only problem is that UML painters and code crunchers usually apply it in every little step, meaning there is premature optimization all over the project.

The following set of heuristics is a correction:

1. Make it work.
2. Make it right (the code is readable [uses IntentionRevealingNames] and every idea is expressed OnceAndOnlyOnce).
3. Make everything work.
4. Make everything right.
5. Use the system and find performance bottlenecks.
6. Use a profiler in those bottlenecks to determine what needs to be optimized. See ProfileBeforeOptimizing.
7. Make it fast. You maintained unit tests, right? Then you can refactor the code mercilessly in order to improve the performance.

-- GuillermoSchwarz

I'm not sure you can't know where your bottlenecks are going to be. Don't you guys do order estimates before you code? -- Anonymous

As in BigOh order? You could, but those are sensitive to data set size and other factors that you might not know ahead of time, or that could change over time during development. A team could expend much effort coding an nlogn solution in the belief that the customer's assertion that the data is enormous or the performance factor is critical, only to find out in acceptance tests that the customer's idea of 'enormous' is laughably small. If it does turn out to be a suboptimal algorithm, BigOh can tell you where you are and what your choices for better solutions might be. Better to get a correct solution as outlined above, and then when performance bottlenecks are found, use BigOh to determine if the bottleneck is genuinely caused by that order n^2 code or by the fact that the data is being read over NFS instead of off a local disk. -- StevenNewton

This was originally attributed to "CarHoare" above, but tracking down the origin of this quote I found that it was actually Knuth who said it first. Although Knuth did calls it "Hoare's dictum" 15 years later, Hoare himself disclaimed it. See http://shreevatsa.wordpress.com/2008/05/16/premature-optimization-is-the-root-of-all-evil/ I think we can assume, with fair certainty, that Knuth was the first to use the phrase, and that it passed into folkore subsequently, leading to the wrong people being credited. Does anyone have any further information? -- ShreevatsaR

Absolute statements are the root of all evil. If you optimize too much upfront, you may have wasted time on something that was unnecessary. If you don't optimize upfront, you may waste time later reorganizing a lot of code or the customer may suffer in silence (realizing it's often too late to complain once the produce is "finished".)

The same applies to design in general. If you design too much upfront, you may have wasted time abstracting cases that will never be used. If you don't design upfront, you may waste time later reorganizing...

The key is balance. There are no answers, only questions.

We all become what we do. Do junk and you will only be a good junk producer. Do marvels and you will become better and better at it.

*I have to add a caveat: junk makers are better at handling crises because they have so much experience with crisis due to the fact they create so many. They can better handle an "organic" environment. I once got stuck with a BigBallOfMud and was marked down on my evaluation as being "inflexible". Complaints about poor normalization, no modularization, lack of documentation, and 1965-BASIC-style variable naming ("A2") didn't change my grade. If I built more crap like that I'd be better at handling crap like that.* -top

It takes quite a while to learn how to do well. But it pays also.

If you have enough grey matter, that's a no-brainer: do it as well as possible. <-- oxymoron?

After a month in production, I recently finished un-breaking a process I rewrote from scratch for my company. Despite painstaking efforts to replicate the production environment elsewhere, we couldn't reproduce it in any environment but production. At wits' end, I finally got approval to run production data through the system on my development workstation. Lo and behold, the same symptoms manifested plain as day. I profiled the application and traced the CPU load to an XSL transformation.

As it turns out, early on in the design I had thought it would be a good idea to execute XSL transforms in parallel, because these were CPU bound and we had 4 cores in production. So I had thrown the transformation into a thread pool. Unfortunately, the specific configuration of data in production somehow caused the thread pool to run out of control, overloading the CPU and causing timeouts leading to data loss. I dialed the thread pool down to a single thread, and everything just worked.

I now have a post-it note on my monitor that says, in all caps, OptimizeLater!

## Optimize Later

YouArentGonnaNeedIt applied to Optimization. -- FalkBruegmann

In other words, you are unlikely to know up front whether an optimisation will be of any real benefit. Just write the code the simplest way. If, eventually after profiling you discover a bottleneck optimise that. [ProfileBeforeOptimizing]

Here's a two-page article by MartinFowler: **[next] Code And Then Optimize**

Code written in assembler or C is almost impossible to maintain. Code written in scripting languages is dog-slow. But you can combine the two, and you can profile the dog-slow scripts to find out where the bottlenecks are.

**Therefore**, Don't code for performance. Don't use a "fast" language. Code for maintainability and use a language that improves that maintainability. Then profile your code, find out where the bottlenecks are, and replace only those bits with performance-coded fast-language stuff. The result is that your code will effectively run just as fast as if you'd optimized all of it, but it'll be vastly more maintainable. [AlternateHardAndSoftLayers]

["*Code written in assembler or C is almost impossible to maintain."* I'm sorry, but this is unmitigated bullshit [EditHint: please rephrase with cooler head]. I've been dropped into countless lines of C, C++, and assembly for a couple dozen different processors over the last 35 years, and properly-coded software can be maintained regardless of the level of source being used. Let's please don't get carried away with the hyperbole, shall we? -- MartySchrader]

How much extra effort does it take to develop this "*properly coded software*"? Does it require SelfDiscipline and BestPractices to stray from a PathOfLeastResistance that would not be required in the scripting language? It is empirically proven that developing and maintaining code in traditional 'hard layer' languages can easily be an order of magnitude less productive than doing the same in a scripting language, especially for certain classes of tasks: configuration, animation, UI layout, character AI, and other domains where we need a lot of fuzzy, specialized code with empirical tweaking. Assuming time or budget constraints, an order of magnitude can indeed qualify as "almost impossible". So how much is hyperbole, and how much of ItDepends on context?

Make it work,

make it correct,

make it fast,

make it cheap.

-- (attributed to) AlanKay

And it is way harder than it might seem because:

Make it work (and you will have bugs),

make it correct ( or just GoodEnough ?),

make it fast (and for that, you will modify the code, and add bugs, and that will send you back to Make it work)

make it cheap. (which is very hard to achieve after all the iterations you required to reach this point)

What about making it "safe" (secure, NotEasyToCrack?)? does it go inside "correct"?

OptimizeLater is one of the ways to produce SelfDocumentingCode.

Don't optimize until you need to.

**Rationale**:

There are many reasons to leave optimizations until later. The prime candidate is that the necessity to optimize is a very powerful force when writing the code. If it is applied too early, it can impact the codebase so powerfully that the code becomes unreadable from early on. If code is unreadable, it is unmaintainable, and it is unlikely it will function.

Optimizations almost always require extensive documentation! They are rarely the simplest approach. (cf. YouArentGonnaNeedIt, DoTheSimplestThingThatCouldPossiblyWork, LazyProgrammer).

Also, without accurate information as to what need to be optimized, you are likely to throw resources in the wrong places. TheFireIsWhereItIsHot, not everywhere.

Fast code is easy to write, as long as it it doesn't have any other good qualities. The temptation in early optimization is to ignore other things that matter more.

**Arguments**:

"I like tight, efficient code."'

So do I, I just hate reading it. Moreover, it is difficult to understand, even by the original programmer, and thus highly errorprone.

"Optimized GoodTightCode is elegant!"

No, elegant code is elegant. Optimized code is usually ugly. Optimal algorithms are usually elegant for the sheer genius of them, but their implementations are horrendous.

*But code that is elegant and fast is better than code that is elegant and slow.*

**Exceptions**:

When it comes down to it, you might need to optimize in certain locations. If you had bothered to NarrowTheInterface, this would be easy because you'd only have to change a small locality.

Never write code with the intent to optimize it!

Never say never. The practicalities of the ComputerGamesIndustry mean that I am constantly writing code with the intent to optimize it (later). Many specific pieces of code in a game (e.g. lighting calculations) are executed thousands of times - and this is known in advance. OF COURSE I'm going to write code with the intent to optimize it (later). The statement above is terse enough that I may be misinterpreting it completely, of course! Can someone clarify? -- EddieEdwards

I'm with Eddie on this one. In fact, I'll take it a step further. I program DSPs for a living, and the processing unit I use most has 1K of instruction space and a similar dearth of data space. Note: This is a good reason to optimize (now). I'm constantly fighting a resource battle; at any given time, all of the resources are used and I need to cram in yet another feature. What this means is that I have to further optimize the existing code to make space and then jam in the code for the new feature. This means optimizing (now). If I don't optimize (now), I can't even test the code, much less release it -Timdog

See also DoTheThingThatMightWorkWell, in other words, don't necessarily fully optimize upfront, but certain upfront optimizations are very healthy. Isn't this page (and all related) a jump from one extreme to another? – CostinCozianu

I think the above can be summarized by saying "OptimizeLater" - unless you have worked in this ProblemSpace before, using these technologies, in which case you use HeuristicRules of particular optimizations that you know apply to the environment. – RobertMcAuliffe

Optimize Later is not an excuse for initial incompetence. Programmers should have a clear understanding of what affects application performance. It is a required skill. They should keep this in the backs of their minds while coding, but without it being a driving force. For example, you should know if you have a long list of items that need to be looked up by a key then you might need to use an indexed Set or Map of some sort, and not an array or a Vector. This is not an optimization, it is just doing it right in the first place.

It is an optimsation. If you have correctly factored your code then it doesn't matter what representation you use. Then you can do the simple thing first and optimise it only when you need to. What programmers need to keep in the back of their mind, and the front of their mind, is that this code may, almost certainly will, need to change, so it should be simple, clear, elegant, well-factored, and easy to change. Then it doesn't matter if the first choices are wrong, you can put it right later. Yes, programmers should have a clear understanding of what affects performance, but that just means even more that they should correctly modularise and factor their code. Ability to change when appropriate is what really matters.

This is true, and illustrates the point. If code is well factored, the cost difference between using one solution that is slow and one solution that is faster (but not necessarily optimal) can be minimal, if made up front. Often, it's just a question of selecting a suitable class, method or algorithm, rather than selecting an unsuitable one, in circumstances where the choice is clear and simple. But changing it later, even if the code is well written is always expensive. You have to find the problem, you have to change it, and you have to retest it. Furthermore, if a programmer makes the wrong choice once, they are very likely to make it again.

You have confused me - you seem to be arguing both sides at once. My point is that if you have properly factored code then changing you class, method or algorithm will not be expensive. Work done now is speculative. Optimisation done now may not have been necessary. Many people have optimised idle loops. Don't optimise until you have profiled and know that it's needed at this spot. If your code is well written it won't be more expensive to change it later.

Yes, I know the caveats and exceptions, yes I know this is not universally true, but you will save more time by using this guiding principle than it will cost you.

Where this issue might be important is, for example, in a code review. If a reviewer sees that a trivial change can be made that will clearly improve the performance of the system without affecting the clarity or functionality of the code, should they insist the change be made? If the change is made, the cost may be a few minutes of time. In addition, the developer whose code is being reviewed may learn something. In the future, the developer may now deliver more efficient code at zero cost. If the code is not changed, there is a chance that it may have to be fixed later, at a much higher cost. Furthermore, the developer may introduce further inefficiencies in the system, which cumulatively could have a major effect.

Over-optimization is bad, but that does not mean that considerations of efficiency have no value. They have value, but they are subservient to those of structure. Where the structure is the same, the more efficient solution is the right one.

**Examples**:

Text searching is a very well studied field. There are many good algorithms for doing fast text searches beyond the simple loop and test (aka scan) algorithm. However, in most cases, a simple loop and test (or even better, strcmp() or string::operator ==) are likely to get you where you want to go with the minimum of hassle. If you're writing a web search engine, well maybe you need a better search algorithm. But keep that contained in a nice module so it can be changed, upgraded and tested easily.

A long time ago, people suggested to me that instead of

y = x \* 320;

I should write

// y = x \* 320;

y = (x << 8) + (x << 6);

(or even worse, without the comment)

The theory is that bitshifting is faster than multiplication. However, I figure, if I'm going to multiply by 320, I should multiply by 320. Moreover, most modern compilers will actually optimize the multiplication to be the shifts anyway and are, in fact, faster at it than you are!

Not to mention the fact that you shouldn't be using hardcoded values like that anyway. NamedConstants are much better.

Actually, as it turns out, most modern (embedded systems) compilers won't change the multiplication to bit shifts for you. Yes, I recognize the irony in contradicting my own anecdote albeit years later. Perhaps there's a lesson in that. -- SunirShah

Also, that bit shifting could be slower on most systems. If a CPU can execute a multiplication instruction in one clock cycle then your bit shifting instruction may take 3 cycles (shift, shift, and add). And if you are developing this on a microcontroller with limited instruction space, your going to end up using more instructions to accomplish the same task. – BlakeMason

Most microcontrollers that are so silicon-starved as to not support a large instruction space will also lack multiply instructions. The most you can hope for is a multiply-step instruction. Shifting is still preferred there too. --Samuel A. Falvo II

Actually, on a Pentium, the multiply operation is a non-parallelizable 3-cycle integer multiply (and two loads, one maybe immediate, and maybe a store) while the shift-and-add operation is 3 possibly parallelizable 1-cycle integer add/shift operations (and three loads and maybe a store). The worst-case run time of the multiply operation is equal to the best-case run time of the add/shift operations, and both of them will either stall or occupy one of the two instruction pipelines. On machines with slow memory (i.e. most modern consumer machines), extra instruction decodes and memory accesses cost much more than even a floating-point multiply.

Most modern CPU's (even those made by Intel) are better than Pentiums - the integer multiply is as fast or much faster than equivalent shifts and adds, except for trivial cases which are equivalent to simple bit shifts (no addition), and of course multiplication by constant 1 and 0.

Microcontrollers without useful integer multiply instructions have no choice but to use the shift/add algorithm, call a function in a math library, or even use application-specific code.

I think this thread of discussion illustrates perfectly why you should start with the FirstRuleOfOptimization, and only then, maybe, move on to OptimizeLater: while effecting multiplication via bitshifting may be faster **on some platforms**, it is certainly not faster on all platforms. Writing your code at the highest semantic level appropriate for the application allows the compiler the greatest amount of leeway (usually) in optimizations, especially when it comes to cross-platform code that may run faster with multiplication or may run faster with bitshifts, and which may be optimized for speed or may be optimized for space. The general rule I follow when considering optimizing my code is that the people who write my compiler are Smarter Than I Am, and know my hardware Better Than I Do. While I’m here, I’ll note that while shifting may outperform multiplication on some systems, there comes a point (in terms of the number of high bits in your multiplicand, and thus necessary shifts and sums) on just about any system beyond which shifting and adding simply will not be faster.

"Optimizations always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught." (from LarryWall)

**Horror Stories from the Trench:**

Once, I met a programmer who attempted to optimize code while typing it in for the first time. He never got his code working. In the end, we had to chuck both the code and him and start again. -- SunirShah

I have just built an elegant and extremely slow GUI program. I'll let you know next week whether I optimize it into performing beautifully, or whether my colleagues chuck me and my code and start again. ;-)

Also the bit above about shifting vs multiplication reminds me of an classic war story from someone else's trench: many commonly used compilers were "optimizing" divisions into arithmetic shifts, e.g x div 2 -> x asr 1. This is of course incorrect because when you arithmetic shift right a 2's complement negative number, 1's are placed on the left hand side, so -1 asr n is still -1! This is described in the paper ArithmeticShiftingConsideredHarmful. – LukeGorrie

Actually, "still -1" is the right answer! The division gives -0.5, which rounds to -1 for an integer result.

Most definitions of integer division specify that the result is truncated, not rounded (or floor()ed): -1/2==1/2==0.

It depends on the programming language. Some languages, like C and Java, have division that rounds towards 0. Other languages, like Python and Ruby, have division that always rounds down. In those languages, division by 2 and right shift by 1 would be equivalent.

I was working with a Unix/Oracle business system that was running too slow. A highly paid "expert performance consultant" took exception to the code at the start and end of each function that said "if global\_debug\_flag is set, then call trace routine." Consultant demanded that we delete all the "if debug" statements, due to performance issues. (Not make them a conditional compile; DELETE them!) (Numbers: Maybe 200 flag tests a transaction, all being false. Each transaction also does about two dozen database calls.) I insisted he was full of it, and refused to do it.

Soon after that, we made the program stop inserting a record into a certain empty table at the start of each transaction, and deleting the record at the end of each transaction, leaving the table empty. That made the program run 3 times faster.

(In Oracle, the overhead to add the first extent to a table, or drop the last one, is rather high. ;-) – JeffGrigg

Early **dBASE-II** advertisements described how their product was unusually fast and efficient because it was "*written entirely in assembly language."*

However, as anyone who actually used it could tell you, it was dog slow:

* Could only be programmed through an inefficient interpreted language that was not even tokenized in any way. (No compiler, compiled language, or interfaces from/to compiled languages.)
* O(N) insertion into laughably simplistic index files. (So population of large tables / "sorting" was O(N^2).)

*(...not to mention being chock full of bugs! The 16-bit version, in converted assembly, was DOA -- practically unusable.)*

I actually used it, and it was lightning fast, compared to alternatives available to the average users at the time (early 1980's). You could run scripts and the scripts could be compiled and used via Clipper! You could be done in the time Windows takes to boot up! And if you take the time it takes Access to load you could be into the second thing for the day, instead of still waiting for the program to get ready!

PrematureOptimization often makes things slower anyway. This is because optimizations work against modularity; they use one's global knowledge of the program to cut across modularization and produce a solution that is faster overall, but less easy to understand, since the new solution combines details of how several parts operate, in order to produce a special-case algorithm that couples them together. But many systems are too slow not because the algorithm at their core is slow, but because there is insufficient modularization and poor structuring, and so unintentional duplication of work.

I'm surprised not to find on this page or elsewhere my golden rule for any optimisation, which is to define it such that it can be switched on or off with a simple flag. You then turn the flag on or off in the code, with start-up switches or even via a debug menu. – DavidWright

It seems this rule would lead to a combinatoric explosion of flags, plus you now must test and maintain both the original and optimised versions of each optimisation.

*I do see one way of making this work in a maintainable manner: define optimizations as 'code transforming' stages, annotate code (e.g. with identity functions) to 'suggest' points of profitable transformation, then use switches to enable/disable these passes. This would be similar to how the LLVM optimizer works. However, it is more difficult to develop generic optimizations that would work this way.*

One problem is that this very good advice often leads developers to make the step from OptimizeLater to skip optimization altogether resulting in slow applications and servers. Often as developers, things that we can do later often get skipped completely in a crunch. So by all means optimize later and definitely optimize what matters, but always measure so you don't introduce gates. Optimize what matters is really important, for example there is usually no gain from optimizing some loop conditions if the loop is performing database queries.

Jonathan Blow, the creator of Braid recently talked about becoming an independent game programmer, http://the-witness.net/news/2011/06/how-to-program-independent-games/ and made two exceptionally good points:

1. That we seldom know what is really slow in code (underlining the focus on profiling over optimization)
2. If you're optimizing holistically, you should also be optimizing for "programmer time spent": heavily and unnecessarily optimized solutions and their effect on code maintainability and extensibility make the overall process far less optimal.

## Yet Another Optimization Article

Martin Fowler *(2002, ieee mag)*

This is a troubling column to write. I hadn’t planned to write on optimization, because what I have to say has already been said numerous times. Yet, when I give talks, I find there’s still a surprising number of people who don’t know, or at least don’t follow, the advice I’m about to give. So, here goes. (Many of you have probably seen this ad vice before—my thought to you is to ponder why I need to say this again.)

First, performance matters. Al though relying on Moore’s law to get us out of slow programs has its merits, I find it increasingly annoying when I get a new version of a program and must upgrade my hardware for it to work acceptably. The question is, “How do we achieve a fast program?”

For many programmers, performance is something you pay continuous attention to as you program. Every time you write a fragment of code, you consider the performance implications and code the program to maximize performance. This is an obvious technique—pity it works so badly.

Performance is not something you can work on in this way. It involves specific discipline. Some performance work comes from architectural decisions, some from a more tactical optimization activity. But what both share is the fact that it is difficult to make decisions about performance from just looking at the design. Rather, you have to actually run the code and measure performance.

**Steps for optimization**

Optimizing an existing program follows a specific set of steps. First, you need a pro filer—a program that can analyze how much time your program spends in its various parts. Software performance has an 80/20 rule: 80 percent of the program’s time is spent on about 20 percent of the code. Trying to optimize performance in the 80 percent of code is futile, so the first order of business is to find that 20 percent of code. Trying to deduce where the program will spend its time is also futile. I know plenty of experienced programmers who always get this wrong. You have to use a profiler.

To give the profiler something to chew on, perform some kind of automated run that reasonably simulates the program under its usual conditions. An automated test suite is a good starting point, but make sure you simulate the actual conditions. My colleague Dave Rice has a rule: “Never optimize a multiuser system with single-user tests.” Experience has taught us that a multiuser database system has very different bottlenecks than a single user system—often focused around transaction interactions. The wrong set of tests can easily lead you to the wrong 20 percent of code.

Once you’ve found your bottlenecks, you have two choices: speed up the slow things or do the slow things less often. In either case, you must change the software. This is where having a well-designed piece of software really helps. It’s much easier to optimize cohesive, loosely coupled modules. Breaking down a system into many small pieces lets you narrow down the bottlenecks. Having a good automated test suite makes it easier to spot bugs that might slip in during optimization.

It’s worth knowing about various optimization tricks, many of which are particular to specific languages and environments. The most important thing to remember is that the tricks are not guaranteed to work—as the saying goes, “Measure twice, cut once.” Unlike a tailor, however, you measure before and after you’ve applied the optimization. Only then do you know if it’s had any effect. It’s revealing how often an optimization has little—or even a negative—effect. If you make an optimization and don’t measure to confirm the performance increase, all you know for certain is that you’ve made your code harder to read.

This double measurement is all the more important these days. With optimizing compilers and smart virtual machines, many of the standard optimizing techniques are not just ineffective but also counterintuitive. Craig Larman really brought this home when he told me about some comments he received after a talk at JavaOne about optimization in Java. One builder of an optimizing virtual machine said, in effect, “The comments about thread pools were good, but you shouldn’t use object pools because they will slow down our VM.” Then another VM builder said, “The comments about object pools were good, but you shouldn’t use thread pools because they slow down our VM.” Not only does this reinforce the need to measure with every optimization change, it also suggests that you should log every change made for optimization (a comment tag in the source code is a good option) and retest your optimizations after upgrading your compiler or VM. The optimization you did six months ago could be your bottleneck today.

All of this reinforces the key rule that first you need to make you program clear, well factored, and nicely modular. Only when you’ve done that should you optimize.

**Some exceptions**

Although most performance issues are best dealt with in these kinds of optimizations, at times other forms of thinking are important—for example, during early architectural stages, when you’re making decisions that will be costly to reverse later. Again, the only way to really understand these performance issues is to use measurements. In this case, you build exploratory prototypes to perform crude simulations of the environments with which you’re going to work and to get a sense of the relative speeds. It’s tricky, of course, to get a good idea of what the actual environment might be, but then it’s likely that everything you’re working with will be upgraded before you go live anyway. Experiments are still much better than wild guesses.

There are also some cases where there are broad rules about slow things. An example I always come

across is the slowness of remote procedure calls. Because remote calls are orders of magnitude slower than in-process calls, it’s important to minimize them, which greatly affects overall design. However, this doesn’t trump measuring. I once came across a situation where people optimize remote methods only to find their bottlenecks were elsewhere. However, minimizing remote calls has proven to be a solid rule of thumb.

f you make an optimization and don’t measure to confirm the performance increase, all you know for certain is that you’ve made your code harder to read.

Some have taken this further, coming up with performance models that you can use to assess different architectural designs. Although this can be handy, it’s difficult to take it too far. It all depends on how good the performance model is, and people usually cannot predict the key factors, even at a broad level. Again, the only real arbiter is measurement.

In the end, however, performance is not an absolute. Getting a program to run faster costs money, and it’s a

business decision whether to invest in a quicker program. One value of an explicit optimization phase is that the cost of getting fast performance is more explicit, so businesses can trade it against time to market or more features. Much of the reason why I curse at slow software is because it makes good business sense for the builder.

As I’ve said, much of this advice—in particular, the advice to write a good clean program first and optimize it later—is well worn. (For a longer description of this approach, read Chapters 28 and 29 of Steve McConnell’s Code Complete, Microsoft Press, 1993.) Good quotes about the perils of premature optimization have been around for over 20 years. The pity is that some people still object when I call the same query method twice in a routine. For everyone who finds nothing new in this column, there exists another challenge—how to make it so there is no need to rewrite it in 10 years.

## C2: Rules Of Optimization

The "rules" of optimising are a rhetorical device intended to dissuade novice programmers from cluttering up their programs with vain attempts at writing optimal code. They are:

* FirstRuleOfOptimization - Don't.
* SecondRuleOfOptimization - Don't... yet.
* ProfileBeforeOptimizing

It is uncertain at present, whether cute devices such as this have, or ever will, change any attitudes.

*It changed mine.*

*Mine, too.*

Source: MichaelJackson used to say (when asked about optimization):

1. Don't.
2. Don't Yet (for experts only).

This is republished in JonBentley's ProgrammingPearls.

And lets not forget these famous quotes:

*"The best is the enemy of the good." -- MrVoltaire*

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity."

-- W.A. Wulf

"We should forget about small efficiencies, say about 97% of the time: PrematureOptimization is the root of all evil." -- DonKnuth (who attributed the observation to CarHoare)

## C2: Premature Anything

If it's *premature*, then it's too early for whatever it is you have in mind (or other body part). So, don't do it (if you're able to not).

Instead of optimizing your code before it's written, try taking a cold shower. Instead of generalizing on a concept before observing even a single instance of it, try getting some sleep.

Sometimes words get left out of phrases in the interest of brevity. Well, brevity's never been my problem, so I like to bring 'em back. Sometimes when we say premature X we mean premature attempt to X, which is a bit different. PrematureOptimization can't exist, unless we want things to be bad. Premature attempts to optimize fail and leave a wake of damage. PrematureGeneralization and PrematureStandardization may or may not imply their literal interpretations. You may succeed in generalizing and be sorry for it. In that case, both the attempt and the success were premature.

But usually we mean "you will fail if you try to do this now", not "you will succeed and be sorry you did if you try this now". (I thought about that for a long time.) -- WaldenMathews Last edit June 2002

## C2: Profile Before Optimizing

All other things being equal, everyone wants their code to run as fast as possible.

An unceasing temptation is to "optimize as you go", by writing things at a lower level than you really should (e.g. reaching directly into arrays, using a reference to an instance variable in an overridable method instead of using the getter method, etc.) or adding lots of execution shortcuts for special cases.

**This almost never works.**

Human beings, even experienced programmers, are very poor at predicting (guessing) where a computation will bog down.

**Therefore**:

Write code according to constraints besides performance (clarity, flexibility, brevity). Then, after the code is actually written:

1. See if you actually need to speed it up.
2. Profile the code to see where it's actually spending its time.
3. Focus on the few high-payoff areas and leave the rest alone.

There are lots of ways to improve performance once you know where to do it: use a data structure that better fits your needs (lots of inserts vs. lots of deletes, lots of space vs. lots of time, etc.), make your algorithm more clever (cache results, take advantage of order, traverse only what you need to, etc.), switch to a lower-level language, or even implement the high-payoff area in hardware.

But if you start haphazardly trying to optimize before you actually know where things are bogging down, you're guaranteed to be pessimizing your development efficiency.

When it comes time to optimize a piece of software, always get profiling information first, so you can tell where you need to spend your time making improvements. Without profiling data, there is no way of knowing for certain whether any "improvements" have indeed improved the code (very similar to using UnitTests to determine when a project is finished).

Usually, "profiling" means getting a profile of time spent in various routines or subsystems. This allows for optimizing for speed. Optimizing for [memory] space, or cache misses, or whatever else can be done, though some might take a little wizardry to get good profile data.

Optimizations need not be tiny tweaks, either. They could be wholesale replacement of a O(N\*N\*N) algorithm with a O(N\*N), or outright elimination in the most grievious cases.

*ThreadMode discussion:*

[[ Someone please link in examples, eg Kent's war-story regarding three 12K strings or date-range objects (perhaps somewhere out in NullObject space?) ]]

Does somebody have experience with profiling in Java? I had problems because the JustInTimeCompiler changes the speed behaviour of the program, therefore making it difficult to profile. It seemes that the JustInTimeCompiler is turned off when getting profiling information.

Hm... any ProfilerTool which measures speed of a program requires clock cycles and memory, I suppose. HeisenbergUncertaintyPrinciple, eh? Observing something changes it. ^\_^ So fudge a few tenths of a second? See JavaProfilers

I generally agree with this - but how does this idea interact with some of the standard performance idioms you use as a matter of course? For example, in C++ I frequently pass parameters by const reference, not value, so as to avoid the overhead of frequent copy constructors. Yes, in a sense that is an optimisation, but it doesn't (IMHO) impact maintainability, and can dramatically impact performance. It's also tedious to put in after the fact since it involves lots of edits. -- BurkhardKloss

*Ferinstance, consider k db (from http://www.kx.com, or see http://www.aplusdev.org for its precursor): every operation performed by the machine is a transaction. If you put 50 machines in a cluster they combine to form a single-threaded state machine. But that's very fat state, such as multi-gigabyte database columns being "single objects". And the result is meaningfully fast, with real time performance guarantees. Anyways, the specific relevance here is that k semantics are call by value, but the underlying implementation is call by reference with copy on write. The more general relevance is that here is an example of a very-high-level design and implementation which, at least in its target domain, beats the socks off of analogous implementations which focus on lower-level optimizations. Or, even more generally, the fact that you focus on const parameters as an issue means you're likely missing the big picture.*

*It's interesting to look at the native java drivers for k db. There's a trivial database interface which is much faster than jdbc -- they can be freely downloaded, last I checked. Connection pooling, and other "optimizations" aren't implemented because they slow the system down. jdbc is layered on top of that, for people who can live with the speed penalty that the jdbc design imposes.*

Yeah, I write in Java, and I always pass parameters by const reference :-) . For things like this, the question is, "Why wouldn't you do this?" If, as you point out, there's no reason not to -- including maintainability -- go ahead and do it: it's free.

If there is any impact on clarity or any other aspect of maintainability, though, then you're making a purchase, and you should put off making it until you know whether you need to. The principle is stated extremely because most of us, most of the time, think of things as free that aren't (e.g. external iteration code). –GeorgePaci

I believe this is merely an example of using an language as it is intended, not an optimization. In C++, you need to be aware of the differences between references, pointers, and values and use them appropriately. –WayneMack

When it comes down to it, passing by const reference simply to avoid a copy is an optimization, and more importantly **one the compiler can do for you** (and yes, I have tested this (with gcc 4.7)). One of the nice things about C++ specifically is that it goes to great lengths to promote so-called “value semantics,” the homoiconity between fundamental types and complex user-defined data types that allows you to Say What You Mean. Limiting the use of const reference parameters to things like copy constructors and assignment operators, things for which crefs are absolutely the norm (to the extent that the compiler will provide their definitions for you!), I find makes code appreciably, though not overwhelmingly, more readable. I don’t have to worry about whether I actually can make my functions const or whether I have to call by value, because having proper value semantics ensures that things Just Work. In other words, don’t be so quick to give up the generalizability (read: future extensibility) and possibly clarity of your code to do something which not only may be unnecessary, but may in fact be done for you once you crank your compiler up to -O11! In (yet) other words, ProfileBeforeOptimizing.

BurkhardKloss writes *how does this idea interact with some of the standard performance idioms you use as a matter of course?*

The example he gives turns out not to illustrate his point well, but there are others. In Java, it is common to use a StringBuffer rather than the string concatenation operator (+). And some recommend declaring local variables outside loops rather than in them. Both of these performance idioms reduce readability (unless you are very familiar with them, at which point they are transparent).

Although these tricks apparently helped in Java 1.0, they are both approximately neutral in 1.2, and can be harmful to performance in 1.3. Why? Because the JVMs and compilers have improved substantially and are now clever enough to optimize away the performance issues that the idioms fix. But they aren't clever enough to recognize what they idioms are doing, so they can't apply the same optimizations.

So a former performance idiom has become a liability. And for many programmers who were taught these as rituals rather than optimization tricks, they have become OldRulesWithForgottenReasons. Yuck! So the general rule of ProfileBeforeOptimizing still holds.

-- WilliamPietri

Probably first published in The ElementsOfProgrammingStyle by BrianKernighan and PjPlauger. They have some other maxims in this area:

* Make it clear before you make it fast
* Make it correct before you make it fast

The bottleneck isn't where you think it is.

I can certainly recall examples from my own and colleagues' code where I (or they) did not know where the bottleneck was. However, I can also recall many examples where, after a moment's thought, I did know where it was.

The very fact that you know which were which shows that you tested your intuitions, rather than just assuming they were always right, which illustrates the thesis of this rule. Same is true of the fact that your intuitions weren't correct 100% of the time...that too illustrates the value of the rule. Obviously some people will never have correct insight, but they should only be following the first of the RulesOfOptimization, not the third.

The ProfilerTool is still useful as insurance if nothing else. Your time is valuable. Even if you're right, you need objective evidence to justify spending time fixing the performance problem. So I'm not disputing the advice to ProfileBeforeOptimizing. However, I don't recall ever seeing any evidence that the rule itself ever been tested scientifically. Is it just FolkWisdom? Or is my concern MetaWisdom gone awry?

Even if you know exactly where the bottleneck is, it's often handy to measure the effect of your optimizations, if only so you know when to stop optimizing.

Also, it's nice to have concrete numbers to satisfy your curiosity and to put in your status report.

This is how I profile: I launch the program, then stop it manually a few times. Each time I stop it, I have complete access to all context information - local vars, the stack trace, and everything. You don't need that many samples to know that e.g. a certain SQL execution, or some string copying, or whatnot, is eating a lot of CPU cycles.

At least, this method tells me where the wall clock time goes, which is often the most important. A situation where my method is indispensable (compared to automatic profiling tools) is in interpretive environments. Sure, it spends its time handling some large datastructure, but \_where\_ in that datastructure. So what my method gives me is not just "which function uses the time", but just as important "what are all those data that it is called with, all the time".

The method is also excellent in a distributed system: "Oh, so it is waiting for that server most of the time", "let's see what it was asking for - could we maybe bulk process that?".

Also, my profiling doesn't interfere with the normal execution - or, well, it requires debug information, which rules out certain optimizations, but aside from that... -- BjarkeDahlEbert, Nov 2003

A good ProfilerTool should be able to capture key values for you as it profiles, recording these against the timestamps it collects. You shouldn't need to stop the program manually. This is much less likely to be accurate. For instance, the break you're providing by stopping the program could allow asynchronous database reads to progress outside your measured execution time, skewing the results.

MikeDunlavey replying to the above paragraph: Assume the program is not optimal - it's spending some wall time it doesn't need to, and you want to find that. If there is a small routine doing a bubble sort of a large integer array, then any CPU profiler showing self % should find it. If it is sorting strings, calling strcmp, then only profilers showing inclusive (total) % would show it. If the routine is not small, but is big as many are, you are left hunting in it for the guilty code, possibly focusing on the wrong statements. If the profiler gives you a graph with the inclusive %, maybe you can find it if strcmp is not called in too many places. Now suppose the problem is not a long bubble sort, but a few routines each being called one time too many times from somewhere. If this is happening simultaneously at different levels of the call stack, those factors can multiply together, but no CPU profiler will really show where the problem is. Suppose the problem is various different class objects being created and their constructors being run when they could in fact be re-used. CPU profilers will have a hard time with this because it is not clear how to summarize the results in a way that draws attention to the problem. Now suppose the call tree is fairly deep, going down into library, system, or routines written by others, that just happen to do some IO, like logging something you're not aware of, pinging out for an internet address, looking for a resource on a dll, shelling out some command-line function, etc. etc. If the profiler is a CPU-profiler, it is blind to that, so there is no way it can possibly tell you a big % of wall time is going into that. The long and short of all this is - typically you need to FIND THE PROBLEM, not MEASURE STUFF. Accuracy of measurement is not the goal. If you can find out what's really taking a lot of time that you can fix, do you really care if you only know the amount of time is somewhere between 40% and 60%? If you fix it, you can easily find out how much time you saved, to any accuracy, by just timing it. But if you can't find the problem, you can't fix it. There is a widespread assumption, which is not a theorem, that any of these problems will be found if you only had the right profiler, but that's a giant IF. Bjarke's method can find any of these problems effectively, because it applies all the intelligence of the programmer to understand what's going on, rather than hoping some automatic summarizer will draw attention to it. If you halt it and examine the state 10 times, then any problem whose fixing would save 50% of time will be precisely in evidence on roughly 50% of the samples. On the other hand, if you want to be told precise measurements, and be told there is no evident problem (though there may be one), then use a profiler. Self-deception always feels good.

See ProfilerTool for discussion and comparison of specific profilers.

I have been collecting some basic optimization rules for years and so far have come up with this list:

* It is faster to do something once than to do it 10,000 times.
* Fastest I/O is no I/O (I think this one is from BrianKernighan)
* Only optimize code that is slow

Of course these rules should be applied only if the above rules have been considered. – RichieBielak

JustOneOptimization - fixing the worst of your problems may cause the next 10 to drop off the scope. Therefore, follow the first three rules of optimization, do JustOneOptimization, then start again at rule #1.

I know this one's widely accepted (i.e. not my idea), but it ain't mentioned here yet and I don't know an apposite Knuth/Kernighan/Ritchie/Beck quote to put on its page to sit along with the other three - anyone got one? -- BrianEwins

I've seen quite the opposite. Fix just one, most of the rest of the system crawls.

It is much better to make the code simpler. JustRefactorTheCode: Simple code has almost always good performance. If the code is simple enough, fixing all the performance problems with just one tweak should be easy. –GuillermoSchwarz

[This is simply not true in some programming domains, in particular I am thinking of numerical analysis (or other heavy simulation code). It is precisely in these sorts of domains where complicated code and algorithms may be justified because they can realize huge gains over simple (and often naive) code. The question of whether this justifies the maintenance cost is , of course, application dependant.]

*ReFactoring code has nothing to do with simplicity or complexity of algorithms, merely how the algorithms are expressed. The more complicated the expression of the algorithm becomes, the more difficult it is to identify areas for possible improvement. It also increases the the liklihood of making otherwise obvious mistakes. One reason simple code tends to have better performance is that it is harder to hide bad decisions.*

I like: "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity." – WilliamWulf

Does anyone have a source for the observation that "Human beings, even experienced programmers, are very poor at predicting (guessing) where a computation will bog down." (and that profilers are better)? It's widely known and I'm not disputing it, but it would be nice to know it was more than anecdotal.... TIA, Alan.

Don't forget what may be the simplest, and quite possibly most effective, way to find performance problems. You run the program under a debugger and, while it is doing whatever takes too long, manually halt it with a "pause" key. Make a record of the call stack. Do it again. Compare the call stacks to find common call instructions. Look for these call instructions in the source code to see if they are truly necessary, or could be avoided. If necessary, take additional samples of the call stack until you find such calls. Four or five is typical; I've never needed more than 20. The reason this works is that the call stack exists for the entire time your program is running, and any call instruction that is active on the call stack some percent of the time, say 30%, would save you that much if you could avoid it. Certainly you could try to speed up the routine being called, but it might be a lot easier just not to call it so much. Unfortunately, most profilers summarize at the level of whole functions / methods rather than on the statements / instructions where the calls occur. -- MikeDunlavey

That seems like a wise and experience-driven approach to profiling.

Thanks, and it's good to see that BjarkeDahlEbert above had the same idea. I think the world is profiler-happy at the moment, which is focussing on the tool, not the problem. I've seen people using profilers be pleased to get 20 percent improvement. With this method, whole gobs of stuff can be cut out, and 20 times is not too unusual.

The profiler says the bottleneck is network latency. The client and server are three hours apart. Now what?

*Find a way to use fewer round-trip interactions between the client and the server.*

I'd say to go with an ISP with more direct wiring. 3 light hours (around 21.64 AU according to google) is about the distance from the Sun to Uranus (no jokes please). I am assuming you aren't from SETI and that electrical signals travel at a significant proportion of the speed of light. -- AaronRobson