JULIAN SHUN: So welcome to the second lecture of 6.172, performance engineering of software systems. Today, we're going to be talking about Bentley rules for optimizing work. All right, so work, does anyone know what work means? You're all at MIT, so you should know.

So in terms of computer programming, there's actually a formal definition of work. The work of a program on a particular input is defined to be the sum total of all the operations executed by the program. So it's basically a gross measure of how much stuff the program needs to do.

And the idea of optimizing work is to reduce the amount of stuff that the program needs to do in order to improve the running time of your program, improve its performance. So algorithm design can produce dramatic reductions in the work of a program. For example, if you want to sort an array of elements, you can use a nlogn time QuickSort. Or you can use an n squared time sort, like insertion sort.

So you've probably seen this before in your algorithm courses. And for large enough values of n, a nlogn time sort is going to be much faster than a n squared sort.

So today, I'm not going to be talking about algorithm design. You'll see more of this in other courses here at MIT. And we'll also talk a little bit about algorithm design later on in this semester. We will be talking about many other cool tricks for reducing the work of a program.

But I do want to point out, that reducing the work of our program doesn't automatically translate to a reduction in running time. And this is because of the complex nature of computer hardware. So there's a lot of things going on that aren't captured by this definition of work. There's instruction level parallelism, caching, vectorization, speculation and branch prediction, and so on. And we'll learn about some of these things throughout this semester.

But reducing the work of our program does serve as a good heuristic for reducing the overall running time of a program, at least to a first order. So today, we'll be learning about many ways to reduce the work of your program.

So rules we'll be looking at, we call them Bentley optimization rules, in honor of John Lewis Bentley. So John Lewis Bentley wrote a nice little book back in 1982 called *Writing Efficient Programs.* And inside this book there are various techniques for reducing the work of a computer program. So if you haven't seen this book before, it's very good. So I highly encourage you to read it.

Many of the original rules in Bentley's book had to deal with the vagaries of computer architecture three and a half decades ago. So today, we've created a new set of Bentley rules just dealing with the work of a program. We'll be talking about architecture-specific optimizations later on in the semester. But today, we won't be talking about this.

One cool fact is that John Lewis Bentley is actually my academic great grandfather. So John Bentley was one of Charles Leiseron's academic advisors. Charles Leiserson was Guy Blelloch's academic advisor. And Guy Blelloch, who's a professor at Carnegie Mellon, was my advisor when I was a graduate student at CMU.

So it's a nice little fact. And I had the honor of meeting John Bentley a couple of years ago at a conference. And he told me that he was my academic great grandfather.

[LAUGHING]

Yeah, and Charles is my academic grandfather. And all of Charles's students are my academic aunts and uncles--

[LAUGHING]

--including your T.A. Helen. OK, so here's a list of all the work optimizations that we'll be looking at today. So they're grouped into four categories, data structures, loops, and functions. So there's a list of 22 rules on this slide today. In fact, we'll actually be able to look at all of them today. So today's lecture is going to be structured as a series of many lectures.

And I'm going to be spending one to three slides on each one of these optimizations. All right, so let's start with optimizations for data structures. So first optimization is packing and encoding your data structure.

And the idea of packing is to store more than one data value in a machine word. And the related idea of encoding is to convert data values into a representation that requires fewer bits. So does anyone know why this could possibly reduce the running time of a program? Yes?

AUDIENCE: Need less memory fetches.

JULIAN SHUN: Right, so good answer. The answer was, it might need less memory fetches. And it turns out that that's correct, because computer program spends a lot of time moving stuff around in memory. And if you reduce the number of things that you have to move around in memory, then that's a good heuristic for reducing the running time of your program.

So let's look at an example. Let's say we wanted to encode dates. So let's say we wanted to code this string, September 11, 2018. You can store this using 18 bytes. So you can use one byte per character here.

And this would require more than two double words, because each double word is eight bytes or 64 bits. And you have 18 bytes. You need more than two double words. And you have to move around these words every time you want to manipulate the date. But turns out that you can actually do better than using 18 bytes.

So let's assume that we only want to store years between 4096 BCE and 4096 CE. So there are about 365.25 times 8,192 dates in this range, which is three million approximately. And you can use log base two of three million bits to represent all the dates within this range.

So the notation lg here means log base of two. That's going to be the notation I'll be using in this class. And L-O-G will mean log base 10. So we take the ceiling of log base two or three million, and that gives us 22 bits.

So a good way to remember how to compute the log base two of something, you can remember that the log base two of one million is 20, log base two of 1,000 is 10. And then you can factor this out and then add in log base two of three, rounded up, which is two. So that gives you 22 bits. And that easily fits within one 32-bit words. Now, you only need one word instead of three words, as you did in the original representation.

But with this modified representation, now determining the month of a particular date will take more work, because now you're not explicitly storing the month in your representation. Whereas, with the string representation, you are explicitly storing it at the beginning of the string. So this does take more work, but it requires less space. So any questions so far?

OK, so it turns out that there's another way to store this, which also makes it easy for you to fetch the month, the year, or the day for a particular date. So here, we're going to use the bit fields facilities in C. So we're going to create a struct called date underscore t with three fields, the year, the month, and the date. And the integer after the semicolon specifies how many bits I want to assign to this particular field in the struct.

So this says, I need 13 bits for the year, four bits for the month, and five bits for the day. So the 13 bits for the year is, because I have 8,192 possible years. So I need 13 bits to store that. For the month, I have 12 possible months.

So I need log base two of 12 rounded up, which is four. And then finally, for the day, I need log base two of 31 rounded up, which is five. So in total, this still takes 22 bits.

But now the individual fields can now be accessed much more quickly, than if we had just encoded the three million dates using sequential integers, because now you can just extract a month just by saying whatever you named your struct. You can just say that struct dot month. And that give you the month. Yes?

AUDIENCE: Does C actually store it like that, because I know C++ it makes it finalize. So then you end up taking more space.

JULIAN SHUN: Yeah, so this will actually pad the struct a little bit at the end, yeah. So you actually do require a little bit more than 22 bits. That's a good question. But this representation is much more easy to access, than if you just had encoded the integers as sequential integers.

Another point is that sometimes unpacking and decoding are the optimization, because sometimes it takes a lot of work to encode the values and to extract them. So sometimes you want to actually unpack the values so that they take more space, but they're faster to access. So it depends on your particular application. You might want to do one thing or the other. And the way to figure this out is just to experiment with it.

OK, so any other questions? All right, so the second optimization is data structure augmentation. And the idea here is to add information to a data structure to make common operations do less work, so that they're faster.

And let's look at an example. Let's say we had two singly linked list and we wanted to append them together. And let's say we only stored the head pointer to the list, and then each element in the list has a pointer to the next element in the list.

Now, if you want to spend one list to another list, well, that's going to require you walking down the first list to find the last element, so that you can change the pointer of the last element to point to the beginning of the next list. And this might be very slow if the first list is very long. So does anyone see a way to augment this data structure so that appending two lists can be done much more efficiently? Yes?

AUDIENCE: Store a pointer to the last value.

JULIAN SHUN: Yeah, so the answer is to store a pointer to the last value. And we call that the tail pointer. So now we have two pointers, both the head and the tail. The head points to the beginning of the list. The tail points to the end of the list.

And now you can just append two lists in constant time, because you can access the last element in the list by following the tail pointer. And then now you just change the successor pointer of the last element to point to the head of the second list. And then now you also have to update the tail to point to the end of the second list.

OK, so that's the idea of data structure augmentation. We added a little bit of extra information to the data structure, such that now appending two lists is much more efficient than in the original method, where we only had a head pointer. Questions?

OK, so the next optimization is precomputation. The idea of precomputation is to perform some calculations in advance so as to avoid doing these computations at mission-critical times, to avoid doing them at runtime. So let's say we had a program that needed to use binomial coefficients. And here's a definition of a binomial coefficient.

So it's basically the choose function. So you want to count the number of ways that you can choose k things from a set of n things. And the formula for computing this is, n factorial divided by the product of k factorial and n minus k factorial.

Computing this choose function can actually be quite expensive, because you have to do a lot of multiplications to compute the factorial, even if the final result is not that big, because you have to compute one term in the numerator and then two factorial terms in the denominator. And then you also might run into integer overflow issues, because n factorial grows very fast. It grows super exponentially. It grows like n to the n, which is even faster than two to the n, which is exponential. So doing this computation, you have to be very careful with integer overflow issues.

So one idea to speed up a program that uses these binomials coefficients is to precompute a table of coefficients when you initialize the program, and then just perform table lookup on this precomputed table at runtime when you need the binomial coefficient. So does anyone know what the table that stores binomial coefficients is called? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yea, Pascal's triangles, good. So here is what Pascal's triangle looks like. So on the vertical axis, we have different values of n. And then on the horizontal axis, we have different values of k. And then to get and choose k, you just go to the nth row in the case column and look up that entry.

Pascal's triangle has a nice property, that for every element, it can be computed as a sum of the element directly above it and above it and to the left of it. So here, 56 is the sum of 35 and 21. And this gives us a nice formula to compute the binomial coefficients.

So we first check if n is less than k in this choose function. If n is less than k, then we just return zero, because we're trying to choose more things than there are in a set. If n is equal to zero, then we just return one, because here k must also be equal to zero, since we had the condition n less than k above. And there's one way to choose zero things from a set of zero things.

And then if k is equal to zero, we also return one, because there's only one way to choose zero things from a set of any number of things. You just don't pick anything.

And then finally, we recursively call this choose function. So we call choose of n minus one k minus one. This is essentially the entry above and diagonal to this. And then we add in choose of n minus one k, which is the entry directly above it.

So this is a recursive function for generating this Pascal's triangle. But notice that we're actually still not doing precomputation, because every time we call this choose function, we're making two recursive calls. And this can still be pretty expensive.

So how can we actually precompute this table? So here's some C code for precomputing Pascal's triangle. And let's say we only wanted coefficients up to choose sides of 100. So we initialize matrix of 100 by 100. And then we call this an init choose function.

So first it goes from n equal zero, all the way up to choose size minus one. And then it says, choose n of zero to be one. It also sets choose of n, n to be one. So the first line is, because there's only one way to choose zero things from any number of things. And the second line is, because there's only one way to choose n things from n things, which is just to pick all of them.

And then now we have a second loop, which goes from n equals one, all the way up to choose size minus one. Then first we set choose of zero n to be zero, because here n is-- or k is greater than n. So there's no way to pick more elements from a set of things that is less than the number of things you want to pick.

And then now you loop from k equals one, all the way up to n minus one. And then your apply this recursive formula. So choose of n, k is equal to choose of n minus one, k minus one plus choose of n minus one k.

And then you also set choose of k, n to be zero. So this is basically all of the entries above the diagonal here, where k is greater than n. And then now inside the program whenever we need a binomial coefficient that's less than 100, we can just do table lookup into this table. And we just index and then just choose array.

So does this make sense? Any questions? It's pretty easy so far, right?

So one thing to note is, that we're still computing this table at runtime, because we have to initialize this table at runtime. And if we want to run our program many times, then we have to initialize this table many times. So is there a way to only initialize this table once, even though we might want to run the program many times? Yes?

AUDIENCE: Put in the source code.

JULIAN SHUN: Yeah, so good, so put it in the source code. And so we're going to do compile-time initialization. And if you put the table in the source code, then the compiler will compile this code and generate the table for you that compile time. So now whenever you run it, you don't have to spend time initializing the table. So idea of compile-time initialization is to store the values of constants during compilation and, therefore, saving work at runtime.

So let's say we wanted choose values up to 10. This is the table, the 10 by 10 table storing all of the binomial coefficients up to 10. So if you put this in your source code, now when you run the program, you can just index into this table to get the appropriate constant here. But this table was just a 10 by 10 table. What if you wanted a table of 1,000 by 1,000?

Does anyone actually want to type this in, a table of 1,000 by 1,000? So probably not. So is there any way to get around this? Yes?

AUDIENCE: You could make a program that uses it. And the function will be defined [INAUDIBLE] prints out the zero [INAUDIBLE].

JULIAN SHUN: Yeah, so the answer is to write a program that writes your program for you. And that's called metaprogramming.

So here's a snippet of code that will generate this table for you. So it's going to call this init choose function that we defined before. And then now it's just going to print out C code.

So it's going to print out the declaration of this array choose, followed by a left bracket. And then for each row of the table, we're going to print another left bracket and then print the value of each entry in that row, followed by a right bracket. And we do that for every row. So this will give you the C code. And then now you can just copy and paste this and place it into your source code.

This is a pretty cool technique to get your computer to do work for you. And you're welcome to use this technique in your homeworks and projects if you'd need to generate large tables of constant values. So this is a very good technique to know. So any questions? Yes?

AUDIENCE: Is there a way to write the output other programs to a file, as oppose to having to copy and paste into the source code?

JULIAN SHUN: Yeah, so you can pipe the output of this program to a file. Yes?

AUDIENCE: So are there compiler tools that can-- so we have three processor tools. Is there [INAUDIBLE] processor can do that? We compile the code, run it, and then [INAUDIBLE].

JULIAN SHUN: Yeah, so I think you can write macros to actually generate this table. And then the compiler will run those macros to generate this table for you. Yeah, so you don't actually need to copy and paste it yourself. Yeah?

CHARLES: And you know, you don't have to write it in C. If it's quicker to write with Python, you'd be writing in Python, just put it in the make file for the system you're building. So if it's in the make file, says, well, we're making this thing, first generate the file in the table and now you include that in whatever you're compiling or/and it's just one more step in the process. And for sure, it's generally easier to write these tables with the scripting language like Python than writing them in C. On the other hand, if you need experience writing in C, practice writing in C.

JULIAN SHUN: Right, so as Charles says, you can write your metaprogram using any language. You don't have to write it in C. You can write it in Python if you're more familiar with that. And it's often easier to write it using a scripting language like Python.

OK, so let's look at the next optimization. So we're already gone through a couple of mini lectures already. So congratulations to all of you who are still here.

So the next optimization is caching. The idea of caching is to store results that have been accessed recently, so that you don't need to compute them again in the program. So let's look at an example.

Let's say we wanted to compute the hypotenuse of a right triangle with side lengths A and B. So the formula for computing this is, you take the square root of A times A plus B times B. OK, so turns out that the square root operator is actually a relatively expensive, more expensive than additions and multiplications on modern machines. So you don't want to have to call the square root function if you don't have to.

And one way to avoid doing that is to create a cache. So here I have a cache just storing the previous hypotenuse that I calculated. And I also store the values of A and B that were passed to the function.

And then now when I call the hypotenuse function, I can first check if A is equal to the cached value of A and if B is equal to the cached value of B. And if both of those are true, then I already computed the hypotenuse before. And then I can just return cached of h.

But if it's not in my cache, now I need to actually compute it. So I need to call the square root function. And then I store the result into cached h. And I also store A and B into cached A and cached B respectively. And then finally, I returned cached h.

So this example isn't actually very realistic, because my cache is only a size one. And it's very unlikely, in a program, you're going to repeatedly call some function with the same input arguments. But you can actually make a larger cache. You can make a cache of size 1,000, storing the 1,000 most recently computer hypotenuse values. And then now when you call the hypotenuse function, you can just check if it's in your cache.

Checking the larger cache is going to be more expensive, because there are more values to look at. But they can still save you time overall. And hardware also does caching for you, as we'll talk about later on in the semester. But the point of this optimization is that you can also do caching yourself. You can do it in software. You don't have to let hardware do it for you.

And turns out for this particular program here, actually, it is about 30% faster if you do hit the cache about 2/3 of the time. So it does actually save you time if you do repeatedly compute the same values over and over again. So that's caching. Any questions?

OK, so the next optimization we'll look at is sparsity. The idea of exploiting sparsity, in an input, is to avoid storage and computing on zero elements of that input. And the fastest way to compute on zero is to just not compute on them at all, because we know that any value plus zero is just that original value. And any value times zero is just zero. So why waste a computation doing that when you already know the result?

So let's look at an example. This is matrix-vector multiplication. So we want to multiply a n by n matrix by a n by one vector. We can do dense matrix-vector multiplication by just doing a dot product of each row in the matrix with the column vector. And then that will give us the output vector.

But if you do dense matrix-vector multiplication, that's going to perform n squared or 36, in this example, scalar multiplies. But it turns out, only 14 of these entries in this matrix are zero or are non-zero. So you just wasted work doing the multiplication on the zero elements, because you know that zero times any other element is just zero.

So a better way to do this, is instead of doing the multiplication for every element, you first check if one of the arguments is zero. And if it is zero, then you don't have to actually do the multiplication. But this is still kind of slow, because you still have to do a check for every entry in your matrix, even though many of the entries are zero.

So it's actually a pretty cool data structure that won't actually store these zero entries. And this will speed up your matrix-vector multiplication if your matrix is sparse enough. So let me describe how this data structure works.

It's called compressed sparse row or CSR. There is an analogous representation called compressed sparse column or CSC. But today, I'm just going to talk about CSR.

So we have three arrays. First, we have the rows array. The length of the rows array is just equal to the number of rows in a matrix plus one. And then each entry in the rows array just stores an offset into the columns array or the cols array. And inside the cols array, I'm storing the indices of the non-zero entries in each of the rows.

So if we take row one, for example, we have rows of one is equal to two. That means I start looking at the second entry in the cols array. And then now I have the indices of the non-zero columns in the first row. So it's just one, two, four, and five. These are the indices for the non-zero entries.

And then I have another array called vals. The length of this array is the same as the cols array. And then this array just stores the actual value in these indices here. So the vals array for row one is going to store four, one, five, and nine, because these are the non-zero entries in the first row.

Right, so the rows array just serves as an index into this cols array. So you can basically get the starting index in this cols array for any row just by looking at the entry stored at the corresponding location in the rows array. So for example, row two starts at location six. So it starts here. And you have indices three and five, which are the non-zero indices.

So does anyone know how to compute the length, the number of non-zeros in a row by looking at the rows array? Yes, yes?

AUDIENCE: You go to the rows array and just drag the [INAUDIBLE]

JULIAN SHUN: Right.

AUDIENCE: [INAUDIBLE] the number of elements that are [INAUDIBLE].

JULIAN SHUN: Yeah, so to get the length of a row, you just take the difference between that row's offset and the next row's offset. So we can see that the length of the first row is four, because it's offset is two. And the offset for row two is six. So you just take the difference between those two entries.

We have an additional entry here. So we have the sixth row here, because we want to be able to compute the length of the last row without overflowing in our array. So we just created an additional entry in the rows array for that.

So turns out that this representation will save you space if your matrix is sparse. So the storage required by the CSR format is order n plus nnz, where nnz is the number of non-zeros in your matrix. And the reason why you have n plus nnz, well, you have two arrays here, cols and vals, whose length is equal to the number of non-zeros in the matrix. And then you also have this rows array, whose length is n plus one. So that's why we have n plus nnz.

And if the number of non-zeros is much less than n squared, then this is going to be significantly more compact than the dense matrix representation. However, this isn't always going to be the most compact representation. Does anyone see why? Why might the dense representation sometimes take less space? Yeah? Sorry.

AUDIENCE: Less space or more space?

JULIAN SHUN: Why might the dense representation sometimes take less space?

AUDIENCE: I mean, if you have not many zeros, then you can figure it out n squared plus something else with the sparse created.

JULIAN SHUN: Right. So if you have a relatively dense matrix, then it might take more space than storing it. It might take more space in the CSR representation, because you have these two arrays. So if you take the extreme case where all of the entries are non-zeros, then both of these arrays are going to be of length and squares. So you already have 2n squared there. And then you also need this rows array, which is of length and plus one.

OK, so now I gave you this more compact representation for storing the matrix. So how do we actually do stuff with this representation? So turns out that you can still do matrix-vector multiplication using this compressed sparse row format. And here's the code for doing it.

So we have this struct here, which is the CSR representation. We have the rows array, the cols array, and then the vals array. And then we also have the number of rows, n, and the number of non-zeros, nnz. And then now what we do, we call this SPMV or sparse matrix-vector multiply. We pass in our CSR representation, which is A, and then the input array, which is x. And then we store the result in an output array y.

So first, we loop through all the rows. And then we set y of i to be zero. This is just initialization. And then for each of my rows, I'm going to look at the column indices for the non-zero elements. And I can do that by starting at k equals to rows of i and going up to rows of i plus one.

And then for each one of these entries, I just look up the index, the column index for the non-zero element. And I can do that with cols of k, so let that be j. And then now I know which elements to multiply. I multiply vals of k by x of j. And then now I just add that to y of i. And then after I finish with all of these multiplications and additions, this will give me the same result as if I did the dense matrix-vector multiplication.

So this is actually a pretty cool program. So I encourage you to look at this program offline, to convince yourself that it's actually computing the same thing as the dense matrix-vector multiplication version. So I'm not going to approve this during lecture today. But you can feel free to ask me or any of your TAs after class, if you have questions about this.

And the number of scalar multiplication that you have to do using this code is just going to be nnz, because you're just operating on the non-zero elements. You don't have to touch all of the zero elements. And in contrast, the dense matrix-vector multiply algorithm would take n squared multiplication. So this can be significantly faster for a sparse matrices.

So turns out that you can also use a similar structure to store a sparse static graph. So I assume many of you have seen graphs in your previous courses. See, here's what the sparse graph representation looks like.

So again, we have these arrays. We have these two arrays. We have offsets and edges. The offsets array is analogous to the rows array. And the edges array is analogous to the columns array for the CSR representation.

And then in this offsets array, we store for each vertex where its neighbor's start in this edges array. And then in the edges array, we just write the indices of its neighbor's there.

So let's take vertex one, for example. The offset of vertex one is two. So we know that its outgoing neighbor start at position two in this edges array. And then we see that vertex one has outgoing edges to vertices two, three, and four. And we see in the edges array two, three, four listed there.

And you can also get the degree of each vertex, which is analogous to the length of each row, by taking the difference between consecutive offsets. So here we see that the degree of vertex one is three, because its offset is two. And the offset of vertex two is five.

And it turns out that using this representation, you can run many classic graph algorithms such as breadth-first search and PageRank quite efficiently, especially when the graph is sparse. So this would be much more efficient than using a dense matrix to represent the graph and running these algorithms.

You can also store weights on the edges. And one way to do that is to just create an additional array called weights, whose length is equal to the number of edges in the graph. And then you just store the weights in that array. And this is analogous to the values array in the CSR representation.

But there's actually a more efficient way to store this, if you always need to access the weight whenever you access an edge. And the way to do this is to interleave the weights with the edges, so to store the weight for a particular edge right next to that edge, and create an array of twice number of edges in the graph. And the reason why this is more efficient is, because it gives you improved cache locality.

And we'll talk much more about cache locality later on in this course. But the high-level idea is, that whenever you access an edge, the weight for that edge will also likely to be on the same cache line. So you don't need to go to main memory to access the weight of that edge again.

And later on in the semester we'll actually have a whole lecture on doing optimizations for graph algorithms. And today, I'm just going to talk about one representation of graphs. But we'll talk much more about this later on. Any questions?

OK, so that's it for the data structure optimizations. We still have three more categories of optimizations to go over. So it's a pretty fun lecture. We get to learn about many cool tricks for reducing the work of your program.

So in the next class of optimizations we'll look at is logic, so first thing is constant folding and propagation. The idea of constant folding and propagation is to evaluate constant expressions and substitute the result into further expressions, all at compilation times. You don't have to do it at runtime. So again, let's look at an example.

So here we have this function called orrery. Does anyone know what orrery means? You can look it up on Google.

[LAUGHING]

OK, so an orrery is a model of a solar system. So here we're constructing a digital orrery. And in an orrery we have these whole bunch of different constants. We have the radius, the diameter, the circumference, cross area, surface area, and also the volume.

But if you look at this code, you can notice that actually all of these constants can be defined in compile time once we fix the radius. So here we set the radius to be this constant here, six million, 371,000. I don't know where that constant comes from, by the way. But Charles made these slides, so he probably does.

CHARLES: [INAUDIBLE]

JULIAN SHUN: Sorry?

CHARLES: Radius of the Earth.

JULIAN SHUN: OK, radius of the Earth. Now, the diameter is just twice this radius. The circumference is just pi times the diameter. Cross area is pi times the radius squared.

Surface area is circumference times the diameter. And finally, volume is four times pi times the radius cube divided by three.

So you can actually evaluate all of these two constants at compile time. So with a sufficiently high level of optimization, the compiler will actually evaluate all of these things at compile time. And that's the idea of constant folding and propagation.

It's a good idea to know about this, even though the compiler is pretty good at doing this, because sometimes the compiler won't do it. And in those cases, you can do it yourself. And you can also figure out whether the compiler is actually doing it when you look at the assembly code.

OK, so the next optimization is common subexpression elimination. And the idea here is to avoid computing the same expression multiple times by evaluating the expression once and storing the result for later use. So let's look at this simple four-line program.

We have a equal to b plus c. The we set b equal to a minus d. Then we set c equal to b plus c. And finally, we set d equal to a minus d.

So notice her that the second and the fourth lines are computing the same expression. They're both computing a minus d. And they evaluate to the same thing.

So the idea of common subexpression elimination would be to just substitute the result of the first evaluation into the place where you need it in future line. So here, we still evaluate the first line for a minus d. But now in the second time we need a minus d. We just set the value to b. So now d is equal to b instead of a minus d.

So in this example, the first and the third line, the right hand side of those lines actually look the same. They're both b plus c. Does anyone see why you can't do common subexpression elimination here?

AUDIENCE: b minus changes the second line.

JULIAN SHUN: Yeah, so you can't do common subexpression for the first and the third lines, because the value of b changes in between. So the value of b changes on the second line. So on the third line when you do b plus c, it's not actually computing the same thing as the first evaluation of b plus c.

So again, the compiler is usually smart enough to figure this optimization out. So it will do this optimization for you in your code. But again, it doesn't always do it for you. So it's a good idea to know about this optimization so that you can do this optimization by hand when the compiler doesn't do it for you. Questions so far?

OK, so next, let's look at algebraic identities. The idea of exploiting algebraic identities is to replace more expensive algebraic expressions with equivalent expressions that are cheaper to evaluate. So let's look at an example.

Let's say we have a whole bunch of balls. And we want to detect whether two balls collide with each other. Say, ball has a x-coordinate, a y-coordinate, a z-coordinate, as well as a radius. And the collision test works as follows.

We set d equal to the square root of the sum of the squares of the differences between each of the three coordinates of the two balls. So here, we're taking the square of b1's x-coordinate minus b2's x-coordinate, and then adding the square of b1's y-coordinate minus b2's y-coordinate, and finally, adding the square of b1 z-coordinate minus b2's z-coordinate. And then we take the square root of all of that. And then if the result is less than or equal to the sum of the two radii of the ball, then that means there is a collision, and otherwise, that means there's not a collision.

But it turns out that the square root operator, as I mentioned before, is relatively expensive compared to doing multiplications and additions and subtractions on modern machines. So how can we do this without using the square root operator? Yes.

AUDIENCE: You add the two radii, and the distance is more than the distance between the centers, then you know that they must be overlying.

JULIAN SHUN: Right, so that's actually a good fast path check. I don't think it necessarily always gives you the right answer. Is there another? Yes?

AUDIENCE: You can square the ignition of the radii and compare that instead of taking the square root of [INAUDIBLE].

JULIAN SHUN: Right, right, so the answer is, that you can actually take the square of both sides. So now you don't have to take the square root anymore. So we're going to use the identity that says, that if the square root of u is less than or equal to v exactly when u is less than or equal to v squared. So we're just going to take the square of both sides. And here's the modified code.

So now I don't have this square root anymore on the right hand side when I compute d squared. But instead, I square the sum of the two radii. So this will give you the same answer. However, you do have to be careful with floating point operations, because they don't work exactly in the same way as real numbers.

So some numbers might run into overflow issues or rounding issues. So you do have to be careful if you're using algebraic identities and floating point computations. But the high-level idea is that you can use equivalent algebraic expressions to reduce the work of your program. And we'll come back to this example late on in this lecture when we talk about some other optimizations, such as the fast path optimization, as one of the students pointed out. Yes?

AUDIENCE: Why do you square the sum of these squares [INAUDIBLE]?

JULIAN SHUN: Which? Are you talking about--

AUDIENCE: Yeah.

JULIAN SHUN: --this line? So before we were comparing d.

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah, yeah, OK, is that clear? OK. OK, so the next optimization is short-circuiting. The idea here is, that when we're performing a series of tests, we can actually stop evaluating this series of tests as soon as we know what the answer is So here's an example.

Let's say we have an array, a, containing all non-negative integers. And we want to check if the sum of the values in a exceed some limit. So the simple way to do this is, you just sum up all of the values of the array using a for loop. And then at the end, you check if the total sum is greater than the limit. So using this approach, you always have to look at all the elements in the array.

But there's actually a better way to do this. And the idea here is, that once you know the partial sum exceeds the limit that you're testing against, then you can just return true, because at that point you know that the sum of the elements in the array will exceed the limit, because all of the elements in the array are non-negative. And then if you get all the way to the end of this for loop, that means you didn't exceed this limit. And you can just return false.

So this second program here will usually be faster, if most of the time you exceed the limit pretty early on when you loop through the array. But if you actually end up looking at most of the elements anyways, or even looking at all the elements, this second program will actually be a little bit slower, because you have this additional check inside this for loop that has to be done for every iteration. So when you apply this optimization, you should be aware of whether this will actually be faster or slower, based on the frequency of when you can short-circuit the test. Questions?

OK, and I want to point out that there are short-circuiting logical operators. So if you do double ampersand, that's short-circuiting logical and operator. So if it evaluates the left side to be false, it means that the whole thing has to be false. So it's not even going to evaluate the right side.

And then the double vertical bar is going to be a short-circuiting or. So if it knows that the left side is true, it knows the whole thing has to be true, because or just requires one of the two sides to be true. And it's going to short circuit.

In contrast, if you just have a single ampersand or a single vertical bar, these are not short-circuiting operators. They're going to evaluate both sides of the argument. The single ampersand and single vertical bar turn out to be pretty useful when you're doing bit manipulation. And we'll be talking about these operators more on Thursday's lecture. Yes?

AUDIENCE: So if your program going to send false, if it were to call the function and that function was on the right hand side of an ampersand, would it mean that would never get called, even though-- and you possibly now find out that, the right hand side would crash simply because the left hand side was false?

JULIAN SHUN: Yeah, if you use a double ampersand, then that would be true. Yes?

AUDIENCE: [INAUDIBLE] check [INAUDIBLE] that would cause the cycle of left hand, so that the right hand doesn't get [INAUDIBLE].

JULIAN SHUN: Yeah. I guess one example is, if you might possibly index an array out of balance, you can first check whether you would exceed the limit or be out of bounds. And if so, then you don't actually do the index.

OK, a related idea is to order tests, suss out the tests that are more often successful or earlier. And the ones that are less frequently successful are later in the order, because you want to take advantage of short-circuiting. And similarly, inexpensive tests should precede expensive tests, because if you do the inexpensive tests and your test short-circuit, then you don't have to do the more expensive tests. So here's an example.

Here, we're checking whether a character is whitespace. So character's whitespace, if it's equal to the carriage return character, if it's equal to the tab character, if it's equal to space, or if it's equal to the newline character. So which one of these tests do you think should go at the beginning? Yes?

AUDIENCE: Probably the space.

JULIAN SHUN: Why is that?

AUDIENCE: Oh, I mean [INAUDIBLE]. Well, maybe the newline [INAUDIBLE]. Either of those could be [INAUDIBLE].

JULIAN SHUN: Yeah, yeah, so it turns out that the space and the newline characters appear more frequently than the carriage return. And the tab and the space is the most frequent, because you have a lot of spaces in text. So here I've reordered the test, so that the check for space is first. And then now if you have a character, that's a space. You can just short circuit this test and return true.

Next, the newline character, I have it as a second test, because these are also pretty frequent. You have a newline for every new line in your text. And then less frequent is the tab character, and finally, the carriage return for character isn't that frequently used nowadays. So now with this ordering, the most frequently successful tests are going to appear first.

Notice that this only actually saves you work if the character is a whitespace character. It it's not a whitespace character, than you're going to end up evaluating all of these tests anyways.

OK, so now let's go back to this example of detecting collision of balls. So we're going to look at the idea of creating a fast path. And the idea of creating a fast path is, that there might possibly be a check that will enable you to exit the program early, because you already know what the result is.

And one fast path check for this particular program here is, you can check whether the bounding boxes of the two balls intersect. If you know the bounding boxes of the two balls don't intersect, then you know that the balls cannot collide. If the bounding boxes of the two balls do intersect, well, then you have to do the more expensive test, because that doesn't necessarily mean that the balls will collide.

So here's what the fast path test looks like. We're first going to check whether the bounding boxes intersect. And we can do this by looking at the absolute value of the difference on each of the coordinates and checking if that's greater than the sum of the two radii. And if so, that means that for that particular coordinate the bounding boxes cannot intersect. And therefore, the balls cannot collide. And then we can return false of any one of these tests returned true.

And otherwise, we'll do the more expensive test of comparing d square to the square of the sum of the two radii. And the reason why this is a fast path is, because this test here is actually cheaper to evaluate than this test below. Here, we're just doing subtractions, additions, and comparisons.

And below we're using the square operator, which requires a multiplication. And multiplications are usually more expensive than additions on modern machines. So ideally, if we don't need to do the multiplication, we can avoid it by going through our fast path.

So for this example, it probably isn't worth it to do the fast path check since it's such a small program. But in practice there are many applications and graphics that benefit greatly from doing fast path checks. And the fast path check will greatly improve the performance of these graphics programs.

There's actually another optimization that we can do here. I talked about this optimization couple of slides ago. Does anyone see it? Yes?

AUDIENCE: You can factor out the sum of the radii for [INAUDIBLE].

JULIAN SHUN: Right. So we can apply common subexpression elimination here, because we're computing the sum of the two radii four times. We can actually just compute it once, store it in a variable, and then use it for the subsequent three calls.

And then similarly, when we're taking the difference between each of the coordinates, we're also doing it twice. So again, we can store that in a variable and then just use the result in the second time. Any questions?

OK, so the next idea is to combine tests together. So here, we're going to replace a sequence of tests with just one test or switch statement. So here's an implementation of a full adder. So a full adder is a hardware device that takes us input three bits. And then it returns the carry bit and the sum bit as output.

So here's a table that specifies for every possible input to the full adder of what the output should be. And there are eight possible inputs to the full adder, because it takes three bits. And there are eight possibilities there.

And this program here is going to check all the possibilities. It's first going to check if a is equal to zero. If so, it checks if b is equal to zero. If so, it checks if c is equal to zero.

And if that's true, it returns zero and zero for the two bits. And otherwise, it returns one and zero and so on.

So this is basically a whole bunch of if else statements nested together. Does anyone think this is a good way to write the program? Who thinks this is a bad way to write the program? OK, so most of you think it's a bad way to write the program. And hopefully, I can convince the rest of you who didn't raise your hand.

So here's a better way to write this program. So we're going to replace these multiple if else clauses with a single switch statement. And what we're going to do is, we're going to create this test variable. That is a three-bit variable.

So we're going to place the c bit in the least significant digit. The b bit, we're going to shift it over by one, so in the second least significant digit, and then the a bit in the third least significant digit. And now the value of this test variable is going to range from zero to seven. And then for each possibility, we can just specify what the sum and the carry bits should be. And this requires just a single switch statement, instead of a whole bunch of if else clauses.

There's actually an even better way to do this, for this example, which is to use table lookups. You just precompute all these answers, store it in a table, and then just look it up at runtime. But the idea here is that you can combine multiple tests in a single test. And this not only makes your code cleaner, but it can also improve the performance of your program, because you're not doing so many checks. And you won't have as many branch misses. Yes?

AUDIENCE: Would coming up with logic gates for this be better or no?

JULIAN SHUN: Maybe. Yeah, I mean, I encourage you to see if you can write a faster program for this. All right, so we're done with two categories of optimizations. We still have two more to go.

The third category is going to be about loops. So if we didn't have any loops in our programs, well, there wouldn't be many interesting programs to optimize, because most of our programs wouldn't be very long running. But with loops we can actually optimize these loops and then realize the benefits of performance engineering.

The first loop optimization I want to talk about is hoisting. The goal of hoisting, which is also called loop-invariant code motion, is to avoid recomputing a loop-invariant code each time through the body of a loop. So if you have a for loop where in each iteration are computing the same thing, well, you can actually save work by just computing it once.

So in this example here, I'm looping over an array of length N. And them I'm setting Y of i equal to X of i times the exponential of the square root of pi over two. But this exponential square root of pi over two is actually the same in every iteration. So I don't actually have to compute that every time.

So here's a version of the code that does hoisting. I just move this expression outside of the for loop and stored it in a variable factor. And then now inside the for loop, I just have to multiply by factor. I already computed what this expression is. And this can save running time, because computing the exponential, the square root of pi over two, is actually relatively expensive.

So turns out that for this example, you know, the compiler can probably figure it out and do this hoisting for you. But in some cases, the compiler might not be able to figure it out, especially if these functions here might change throughout the program. So it's a good idea to know what this optimization is, so you can apply it in your code when the compiler doesn't do it for you.

OK, sentinels, so sentinels are special dummy values placed in a data structure to simplify the logic of handling boundary conditions, and in particular the handling of loop exit tests. So here, I, again, have this program that checks whether-- so I have this program that checks whether the sum of the elements in sum array A will overflow if I added all of the elements together. And here, I've specified that all of the elements of A are non-negative.

So how I do this is, in every iteration I'm going to increment some by A of i. And then I'll check if the resulting sum is less than A of i. Does anyone see why this will detect if I had an overflow? Yes?

AUDIENCE: We're a closed algorithm. It's not taking any values.

JULIAN SHUN: Yeah, so if the thing I added in causes an overflow, then the result is going to wrap around. And it's going to be less than the thing I added in. So this is why the check here, that checks whether the sum is less than negative i, will detect an overflow.

OK, so I'm going to do this check in every iteration. If it's true, I'll just return true. And otherwise, I get to the end of this for loop where I just return false.

But here on every iteration, I'm doing two checks. I'm first checking whether I should exit the body of this loop. And then secondly, I'm checking whether the sum is less than A of i. It turns out that I can actually modify this program, so that I only need to do one check in every iteration of the loop.

So here's a modified version of this program. So here, I'm going to assume that I have two additional entries in my array A. So these are A of n and A of n minus one. So I assume I can use these locations. And I'm going to set A of n to be the largest possible 64-bit integer, or INT64 MAX. And I'm going to set A of n plus one to be one.

And then now I'm going to initialize my loop variable i to be zero. And then I'm going to set the sum equal to the first element in A or A of zero. And then now I have this loop that checks whether the sum is greater than or equal to A of i. And if so, I'm going to add A of i plus one to the sum. And then I also increment i.

OK, and this code here does the same thing as a thing on the left, because the only way I'm going to exit this while loop is, if I overflow. And I'll overflow if A of i becomes greater than sum, or if the sum becomes less than A of i, which is what I had in my original program. And then otherwise, I'm going to just increment sum by A of i.

And then this code here is going to eventually overflow, because if the elements in my array A don't cause the program to overflow, I'm going to get to A of n. And A of n is a very large integer. And if I add that to what I have, it's going to cause the program to overflow. And at that point, I'm going to exit this for loop or this while loop.

And then after I exit this loop, I can check why I overflowed. If I overflowed because of sum element of A, then the loop index i is going to be less than n, and I return true. But if I overflowed because I added in this huge integer, well, than i is going to be equal to n. And then I know that the elements of A didn't caused me to overflow, the A of n value here did. So then I just return false.

So does this makes sense? So here in each iteration, I only have to do one check instead of two checks, as in my original code. I only have to check whether the sum is greater than or equal to A of i. Does anyone know why I set A of n plus one equal to one? Yes?

AUDIENCE: If everything else in the array was zero, then you still wouldn't have overflowed. If you had been at 64 max, it would overflow.

JULIAN SHUN: Yeah, so good. So the answer is, because if all of my elements were zero in my original array, that even though I add in this huge integer, it's still not going to overflow. But now when I get to A of n plus one, I'm going to add one to it.

And then that will cause the sum to overflow. And then I can exit there. So this is a deal with the boundary condition when all the entries in my array are zero.

OK, so next, loop unrolling, so loop unrolling attempts to save work by combining several consecutive iterations of a loop into a single iteration. Thereby, reducing the total number of iterations of the loop and consequently the number of times that the instructions that control the loop have to be executed.

So there are two types of loop unrolling. There's full loop unrolling, where I unroll all of the iterations of the for loop, and I just get rid of the control-flow logic entirely. Then there's partial loop unrolling, where I only unroll some of the iterations but not all of the iterations. So I still have some control-flow code in my loop.

So let's first look at full loop unrolling. So here, I have a simple program that just loops for 10 iterations. The fully unrolled loop just looks like the code on the right hand side. I just wrote out all of the lines of code that I have to do in straight-line code, instead of using a for loop. And now I don't need to check on every iteration, whether I need to exit the for loop.

So this is for loop unrolling. This is actually not very common, because most of your loops are going to be much larger than 10. And oftentimes, many of your loop bounds are not going to be determined at compile time. They're determined at runtime. So the compiler can't fully unroll that loop for you.

For small loops like this, the compiler will probably unroll the loop for you. But for larger loops, it actually doesn't benefit to unroll the loop fully, because you're going to have a lot of instructions. And that's going to pollute your instruction cache.

So the more common form of loop unrolling is partial loop unrolling. And here, in this example here, I've unrolled the loop by a factor of four. So I reduce the number of iterations of my for loop by a factor of four. And then inside the body of each iteration I have four instructions.

And then notice, I also changed the logic in the control-flow of my for loops. So now I'm incrementing the variable j by four instead of just by one.

And then since n might not necessarily be divisible by four, I have to deal with the remaining elements. And this is what the second for loop is doing here. It's just dealing with the remaining elements.

And this is the more common form of loop unrolling. So the first benefit of doing this is, that you have fewer checks to the exit condition for the loop, because you only have to do this check every four iterations instead of every iteration. But the second and much bigger benefit is, that it allows more compiler optimizations, because it increases the size of the loop body. And it gives the compiler more freedom to play around with code and to find ways to optimize the performance of that code. So that's usually the bigger benefit.

If you unroll the loop by too much, that actually isn't very good, because now you're going to be pleading your instruction cache. And every time you fetch an instruction, it's likely going to be a miss in your instruction cache. And that's going to decrease the performance of your program. And furthermore, if your loop body is already very big, you don't really get additional improvements from having the compiler do more optimizations, because it already has enough code to work with. So giving it more code doesn't actually give you much there.

OK, so I just said this. The benefits of loop unrolling a lower number of instructions and loop control code. And then it also enables more compiler optimizations. And the second benefit here is usually the much more important benefit. And we'll talk more about compiler optimizations in a couple of lectures.

OK, any questions? OK, so the next optimization is loop fusion. This is also called jamming. And the idea here is to combine multiple loops over the same index range into a single loop, thereby saving the overhead of loop control.

So here, I have two loops. They're both looping from i equal zero, all the way up to n minus one. The first loop, I'm computing the minimum of A of i and B of i and storing the result in C of i. The second loop, I'm computing the maximum of A of i and B of i and storing the result in D of i.

So since these are going over the same index rang, I can fused together the two loops, giving me a single loop that does both of these lines here. And this reduces the overhead of loop control code, because now instead of doing this exit condition check two n times, I only have to do it n times.

This also gives you better cache locality. Again, we'll talk more about cache locality in a future lecture. But at a high level here, what it gives you is, that once you load A of i and B of i into cache, when you compute C of i, it's also going to be in cache when you compute D of i. Whereas, in the original code, when you compute D of i, it's very likely that A of i and B of i are going to be kicked out of cache already, even though you brought it in when you computed C of i. For this example here, again, there's another optimization you can do, common subexpression elimination, since you're computing this expression A of i is less than or equal to B of i twice.

OK, next, let's look at eliminating wasted iterations. The idea of eliminating wasted iterations is to modify the loop bounds to avoid executing loop iterations over essentially empty loop bodies. So here, I have some code to transpose, a matrix.

So I go from i equal zero to n minus one, from j equals zero to n minus one. And then I check if i is greater than j. And if so, I'll swap the entries in A of i, j and A of j, i.

The reason why I have this check here is, because I don't want to do the swap twice. Otherwise, I'll just end up with the same matrix I had before. So I only have to do the swap when i is greater than j.

One disadvantage of this code here is, I still have to loop for n squared iterations, even though only about half of the iterations are actually doing useful work, because about half of the iterations are going to fail this check here, that checks whether i is greater than j. So here's a modified version of the program, where I basically eliminate these wasted iterations. So now I'm going to loop from i equals one to n minus one, and then from j equals zero all the way up to i minus one.

So now instead of going up to n minus one, I'm going just up to i minus one. And that basically puts this check, whether i is greater than j, into the loop control code. And that saves me the extra wasted iterations.

OK, so that's the last optimization on loops. Are there any questions? Yes?

AUDIENCE: Isn't the checks still have [INAUDIBLE]?

JULIAN SHUN: So the check is-- so you still have to do the check in the loop control code. But here, you also had to do it. And now you just don't have to do it again inside the body of the loop. Yes?

AUDIENCE: In some cases, where it might be more complex to do it, is it also [INAUDIBLE] before you optimize it, but it's still going to be fast enough [INAUDIBLE]. Like in the first example, even though the loop is empty, most of the time you'll be able to process [INAUDIBLE] run the instructions.

JULIAN SHUN: Yes, so most of these are going to be branch hits. So it's still going to be pretty fast. But it's going to be even faster if you just don't do that check at all. So I mean, basically you should just text it out in your code to see whether it will give you a runtime improvement.

OK, so last category of optimizations is functions. So first, the idea of inlining is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself. So here, I have a piece of code that's computing the sum of squares of elements in an array A.

And so I have this for loop that in each iteration is calling this square function. And the square function is defined above here. It just does x times x for input argument x.

But it turns out that there is actually some overhead to doing a function call. And the idea here is to just put the body of the function inside the function that's calling it. So instead of calling the square function, I'm just going to create a variable temp. And then I set sum equal to sum plus temp times temp. So now I don't have to do the additional function call.

You don't actually have to do this manually. So if you declare your function to be static inline, then the compiler is going to try to inline this function for you by placing the body of the function inside the code that's calling it. And nowadays, the compiler is pretty good at doing this. So even if you don't declare static inline, the compiler will probably still inline this code for you. But just to make sure, if you want to inline a function, you should declare it as static inline.

And you might ask, why can't you just use a macro to do this? But it turns out, that inline functions nowadays are just as efficient as macros. But they're better structured, because they evaluate all of their arguments. Whereas, macros just do a textual substitution.

So if you have an argument that's very expensive to evaluate, the macro might actually paste that expression multiple times in your code. And if the compiler isn't good enough to do common subexpression elimination, then you've just wasted a lot of work.

OK, so there's one more optimization-- or there's two more optimizations that I'm not going to have time to talk about. But I'm going to post these slides on Learning Modules, so please take a look at them, tail-recursion elimination and coarsening recursion.

So here are a list of most of the roles that we looked at today. There are two of the function optimizations I didn't get to talk about, please take a look at that offline, and ask your TAs if you have any questions. And some closing advice is, you should avoid premature optimizations.

So all of the things I've talked about today improve the performance of your program. But you first need to make sure that your program is correct. If you have a program that doesn't do the right thing, then it doesn't really benefit you to make it faster.

And to preserve correctness, you should do regression testing, so develop a suite of tests to check the correctness of your program every time you change the program. And as I said before, reducing the work of a program doesn't necessarily decrease its running time. But it's a good heuristic. And finally, the compiler automates many low-level optimizations. And you can look at the assembly code to see whether the compiler did something.