Presentation Title: Programming & Optimizing C Code on Blackfin®

Presenter Name: Alan Anderson

Chapter 1: Concepts and Tools

Subchapter 1a: Introduction

Hi. My name is Alan Anderson, I'm a performance engineer with Analog Devices. The talk today is going to be looking at C performance on the Blackfin processor and understanding how you can improve the performance of C programs. We'll look at the various tools that are available to help you and discuss various scenarios drawn from customer benchmarks that illuminate the situation. There are always a range of techniques available to speed up programs: from engaging automatic optimizations; to intervening in the program; to rewriting parts of the program.

The talk will be broken up into a number of sections, beginning with concepts and tools and understanding the general situation on the Blackfin, then we'll look at tight numeric loops, signal processing kernels vectorization and so on. Then we'll look at what's referred to often as control code, decision-making code, conditional branches and the like. The second to last section touches on Blackfin memory performance, because this can be quite significant today in modern processors. And finally, there is an example that will recapitulate some of the material from a real customer benchmark.

So the first section is introducing some concepts and talking about C and tools available. So, obviously the first question is why you use C? Because the alternative to programming in Hand Assembly is much slower. C enables you to bring in a portable program and quickly experiment with it on the Blackfin and the C is much cheaper to maintain.

Disadvantages are that ANSI C was never designed to be a signal processing language. Its emphasis was on system design rather than mathematics. And DSP processors are often designed to expect hand assembly in certain key areas. And, of course, signal processing applications continue to evolve. They move into new areas constantly, far faster than standard languages like ANSI C can evolve.

So, when you go about increasing the performance of a C program, it's important to realize that the compiler can do certain things, but the programmer has to do certain things as well. Particularly compilers don't intervene at the level of the algorithm. A faster algorithm will almost always be more effective than any compiler optimization. So you work at that level first, considering whether your algorithm is suitable to the facilities on a Blackfin processor. And it's helpful when you're looking at C programs to consider the generality and what we call aliasing, which is to say whether a compiler will be able to understand what data you are accessing at any given time. If you use highly indirect pointers, the compiler will be forced to a conservative strategy.

You have to move beyond the C program to consider the machine's capabilities. Your target will be good at some things. It will have specialized instructions for certain mathematical features like a Viteribi or a bit mux or vectorized multiply and accumulates. You have to consider whether you are going to reach those with the algorithm you've selected.

Lastly in this, you would consider non-portable changes, because up till now we've been talking about still maintaining a C program that will work in any platform. You start to put in non-portable changes when you really want to put in something Blackfin specific to your program. They could be rewrites in C, adding pragmas, or even adding in-line assembly statements. Although we would always advise that you keep a parallel C model for verification. It's important to understand that speeding up a program will occasionally be done by making it more elegant. But more often, it's a specialization of the program for that particular hardware. You will end up with a faster program, which has become larger and more complex and less portable. So in other words, there is a price to pay.

The C language offers what we call the uniform computational model, which means a C programmer can assume his program will give the same answer in any platform. But when we think about performance, you have to realize that the C computational model will be supported in different ways on different platforms. If, for instance, there are native floating point instructions on a machine, then C's assumption that there will be sixty four bit floating point everywhere can be supported efficiently.

But on a machine like the Blackfin, where you don't have native floating point, this will be supported by software emulation and will go considerably slower. So you have to consider this in selecting your algorithm and how you code it. C also assumes a large flat memory model. And, in fact, memory effects are highly significant on modern processors. C is more machine dependent than you might think, even in some simple aspects such as what is the size of a short or an int. If it's 32 or 16 bits.

There can be a poor match with signal processing in the areas of accumulators or vectorization or SIMD and fractional processing. So the somewhat unwelcome message for the C programmer is that the target machine's characteristics will determine your success in terms of performance. That you do have to think down to that level. C programs can be ported with little difficulty, but if you want high efficiency, you really can't ignore the underlying hardware.

I like to split it up into perhaps three different concepts. There is optimization of the program, by which I mean simply letting the compiler do its best, engaging all the optional compiler

optimizations. If you're coming from micro-controller, then you'll be used to an optimizing compiler speeding up your program several times. It can be dramatically more effective in a signal processing loop; where you can find code going twenty times faster, once you optimize it.

But you must note that the optimizing compiler has limited scope in some areas. That it won't change your algorithm and make global changes to your program, or significantly rearrange data, and a compiler will always put correctness ahead of performance. Your job as a programmer after those last few cycles is really to reassure the compiler that it's safe to produce fast code, otherwise it will play conservatively.

The second level is to start changing the program, which we do reluctantly and only as necessary. To elaborate an out-of-the-box portable C program into what I call optimized C. You can do this by adding things like pragmas, which will be treated as comments by other compilers so that you maintain a certain degree of portability. And you can do the same to some extent with built-in functions and memory qualifiers and so on.

And finally, you go to targeted rewrites, to actually amending the program so it really only makes sense on a Blackfin. So we recommend a model to minimize our customer's time to market where you have some idea of your target performance, you come in with your portable C program, and you only go as far down this hierarchy as you need to go to meet your target.

Just to re-emphasize that point about unoptimized versus optimized code, what we see here in the top right is a small single processing loop. Now if you switch on the compiler optimizer, you get the code in the bottom right here, extremely tight code. If you don't switch on the optimizer, you get this code in the left hand panel, which I don't want you to try to strain your eyes to read. But just look at the amount of it. Compared to a microcontroller, it's far easier to understand the optimized code on a Blackfin than it is the unoptimized. So overcome any reluctance to engage the optimization early. It's the best thing to do.

Subchapter 1b: Useful Tools

In speeding up a program, the most important thing is to direct your effort, because any C programmer can open any C program at any page and decide "that looks bad, I should improve that. I can make that go faster." We can do that all day and have no great effect. The important thing is to zoom in on where your program actually spends its time. And that may not be where you expect when you move it to a new hardware target.

The great tool we have for this is statistical profiler. The statistical profiler monitors the execution of your program on our hardware, completely non-intrusively. That is a remarkable advance on older methods of profiling. We used to have to insert tracing code and actually instrument the program which changed the timing characteristics in order to do profiling. But the statistical profiler will simply monitor the program counter as your program runs, and it won't disturb the timing at all. It's completely accurate and it will show timing effects due to I/O memory delay, everything. And you can run it with your normal I/O inputs and outputs as well.

So the key message is you mustn't assume that you already know where your application spends it's time. You must measure it. I can only say that intuition is notoriously bad here. I guess it's the 80/20 rule. 80% of anything is usually noncritical, so you really want to focus your effort on that 20%. Here is a picture from the screen of the statistical profiler running. In the left-hand pane, you'll see a list of functions with the percentage of the time that's spent in each of those functions. If you double click on those, then you find yourself in the right-hand pane which shows similar information only on a line by line basis. So you can very, very quickly zoom in on what are the hot spots for your application.

This kind of facility is also available in the simulator if you're developing on the simulator rather than hardware. Jumping ahead a bit, but while we're talking about the statistical profiler, it is possible to zoom down to the lowest level. If you press the right-hand mouse button when you're hovering over that display of line profiles, you'll find yourself offered something called mixed mode. And mixed mode inserts the machine instructions in amongst the lines of C in your display.

And now we can see the percentage time spent in every single assembly instruction through the program. They may be rearranged a little by the process of optimization. And the tip really is that it can be remarkably effective to just run your eye down these and watch out for instructions that stand out from their neighbors. For instance here we can see that the average time for an instruction is .04 percent of the total. But there we have one taking four times longer. So when you get down to the lowest level of optimizing, you can simply ask yourself - why? What's going on there? Could I rearrange the code so we don't have to do that? In this first case, we're seeing a pipeline stall, and further down here we're seeing the cost of a call instruction, transfers of control taking several cycles.

When you're programming the Blackfin, you have to bear in mind the way in which it attains high speeds is by having a longer pipeline. And that means that when you change the direction of that pipeline with a conditional branch, you'll encounter some stall cycles. So that's going to be significant later as we zoom in on the causes of delays. Similarly, you want to watch out for programming structures like indirection, where the pipe is having to wait to load an address in order to dereference through that address to the next part of the chain.

That is not just inefficient in access time, it's also very hard on the compiler. By the time you get through this chain, the compiler has no idea where the data really is that you're picking up. And consequently no idea if that's conflicting with any other data or the contents of any registers. The C compiler will do its best to get rid of any of these stalls. It will rearrange your instruction sequences to occupy those stall cycles with useful work. But it will be limited in that by how large the piece of code is and whether it understands it well enough to move code around aggressively? There's an engineer to engineer note number 197 which details all these stall conditions.

If you're interested in these stalls and you want to get an overview of what's happening, a useful technique can be the pipeline viewer, which is available in simulation only. What I've done to generate this display is get to a loop of interest in the program. And then simply single stepped the program forward repeatedly and that has populated this display. Now, this looks a bit like overkill. C programmers don't usually want to understand pipelines to this degree of detail. But all you really want to watch for is this slash of yellow. As you single stepped the program forward, these yellow stall cycles will appear in an echelon, and that's all you want to know, that there's a stall situation. If you hover the mouse over those yellow markers and press the control key, up will pop a window that tells you what kind of stall condition you've got.

So, that was the introductory material.

Chapter 2: Tuning DSP Kernels

Subchapter 2a: Tuning techniques

Now we're going to look at more mathematical intensive loops. There is actually no formal difference in C between control code and signal processing code. It's simply a way of breaking up the material. It's important to understand the compiler never knows what kind of material it's processing, it's simply C.

So most programs begin in the signal processing world as floating point, and they are later scaled to transform to fixed point or fractional implementations. We have done a lot of work on the Blackfin to make the floating point support as fast as possible. And you are offered a choice. You

can have what we call strict IEEE floating point and sample performance are shown on the right here in this green panel. Or, if you decide you can live without the Not-a-number or nan-checking that's mandated by strict IEEE, we offer an alternate set of emulation routines which go very much faster. The ratio is shown on the right here of the speed available. And we find very few customers actually need the strict NaN handling, so it's a worthwhile shortcut.

This kind of thing, we find is appreciated by customers in case there are some parts of their application that they can leave in floating point permanently. The Blackfin is a very fast processor and it has been a surprise to everybody to what extent floating point is now left in user applications.

But the next level from floating point is to move to fractional processing. Now the Blackfin instruction set contains a great many operations to offer very fast processing of fixed point and fractional arithmetic, such as the multiply accumulator units and the MACshifts. The compiler and the libraries have a problem getting you to this code, because it's not part of the portable ANSI C definition. So over time we built up a range of different approaches from which you can select the one that suites you best.

We offer fractional builtins, that is to say things that look at like function calls, that take you to specific to fractional arithmetic operators. They map to a single machine instruction in the end. We offer fractional types, fract16, fract32 to help the programming, although those are not native C fractional types. We came across the ETSI operators, they'll be discussed shortly, which have proved to be a useful and portable set of fractional operators. And there is a C++ fractional class, which through operator overloading gives you perhaps the most natural expression of fractional concepts and something that looks like a C program.

And, of course the fractional arithmetic is a hundred times faster than the floating point emulation that we started from. But there is one more way of doing fractions, which generates surprising interest, which is to code the fractions using standard C to spell out the operation using portable C, a language which was never intended for that purpose. So in this loop we see some portable C, which on examination proves to be doing a fractional multiply accumulate, a single machine instruction on the Blackfin.

This kind of representation enables you to maintain complete portability. And the compiler is clever enough to understand this and convert it to the desired instruction. There is some ambiguity in C about what happens when you overflow. In this case the compiler will assume saturation, since that's the normally helpful thing for single processing.

You should note that unsigned variables have rather stricter conditions than signed. So this kind of pattern is more likely to be picked up correctly by the compiler when signed variables are used. There is a problem with accumulation because we don't have a forty bit type. So if you were coding a loop like this that requires scaling, you would scale every time round the loop rather than just at the end.

Of course, if you elaborate this into a number of fractional operators, into a complex expression, you will eventually strain the compiler's ability to recognize what it is you want, which is why you might consider moving to the more explicit forms. So here's a more explicit form. This is what are called fractional built-ins. You'll find these listed in the Blackfin include files. They look like function calls. Here we have one here. But they map to single instructions. The advantage of these is not just that they are explicit to the programmer, but they're explicit to the compiler. The compiler understands exactly what this is and exactly what its side effects are, so it does not in any way disturb the flow of optimization around it.

Here we see listed the ETSI builtins. These were defined by the European Telecommunications Standards Institute, and they were used to specify things like GSM and AMR codecs. But I find that they're supported by quite a number of compiler manufacturers now, so they possibly form a portable way of expressing fractional concepts with precision. These are supported by C programs that show exactly how these would be implemented in portable C, so they have a very clear definition. And we've had many customers very pleased to come along with a standard conforming program, using these, compile it and immediately see themselves approaching their performance target. So these are highly recommended.

Another topic that comes up very often is whether the program should be coded using pointers or arrays. Well it really doesn't make a lot of difference to a modern powerful optimizing compiler. Pointers are sometimes thought to be closer to the hardware, but arrays have the useful property of an implied length. And that helps the compiler very much to understand if two pieces of data overlap, which is always a sticking point in aggressive optimization. If you must choose, try the arrays, but probably it won't matter.

Subchapter 2b: Tight Loops

So, thinking about these tight loops that we're aiming at to do the signal processing. The compiler, until we get to concepts like the profile guided optimizer, the compiler is going to make a rough assumption that you spend most of your time in your program in your inner loops. So it could be a very bad idea to have an inner loop that iterates only once or twice. That would be counter-productive.

The compiler optimizer basically works by unrolling loops. It's looking to stretch out a piece of code so that it has more elbow room to rearrange it, to avoid stalls or to combine operations and vectorizations. The two basic techniques that are used are vectorization and software pipelining. In the second window, you'll see a simple representation of software pipelining. This concept allows work from separate iterations of the loop to be combined in a single machine cycle.

So, for instance, on the third iteration of this loop, we're storing the results of the first iteration; while we do the mathematical, the MAC operation from the second iteration; while we load the data inputs for the third iteration, all in a single cycle. So achieving this pattern is very significant for performance. The reason I'm showing you this is because you need to be able to recognize this pattern to convince yourself if your program has achieved maximum performance or not.

So here it is in a little more detail. The first section is your simply compiled code. We load 16-bit value with another 16-bit value and multiply accumulate them together. On iteration of this little loop taking three cycles. Vectorization would be combining two iterations of this loop. Now our loads are loading 32 bits at a time, two data operands, two 16-bit operands at a time. And here we see a vectorized multiply accumulate. There are two separate MACs. So this has doubled the performance.

Finally we see the effect of software pipelining. Where we load the data, then we enter the loop, and we loop on this single cycle line here, each time doing two multiply accumulates and loading two more sets of data. Finally you exit the loop here at the bottom and you do the final operation. So that has got us to two iterations of the loop in one instruction.

So, basically we're going six times faster. But the trick for the programmer is to check that this has happened. To look at the program and see if these optimizations have taken place, and if they haven't, to ask yourself - why not? What information would the compiler have required? What stopped it basically.

Some advice in these kind of areas is "keep it simple". Don't unroll those inner loops yourself. We often get code coming in that's been used with compilers that haven't been very powerful in

these optimizations and the programmers have rewritten a simple loop, as you see in the top, to something like this. They've basically done the optimization themselves. But on the Blackfin compiler, this is just going to make things worse. This makes it harder for our compiler to recognize the essential situation. We'd far rather see the first piece of code and do all the work in the optimizer. It can, sometimes, be helpful to unroll outer loops.

Very similarly, we'd rather you didn't software pipeline the loops yourself. Here you see a very simple situation; you can image a compiler could understand that. But on the right we see what's happened when it's coming from a device where the programmers felt they had to do the work on the part of the compiler. A little bit of compiler jargon here, I apologize for that.

Things to watch out for: bad things. First, here's an example of a scaler dependency. We have X being set in this expression, and we have X also being read. So this kind of pattern means that the compiler cannot combine iterations. Below we see the same sort of thing coming from the use of an array. We're setting a value in the array, A, but we're also reading. And the index into the array A that we're reading from is itself an array expression, so we have no idea what part of the array A we're reading from. This will stop any of that nice vectorization and software pipelining happening.

Here's a couple of things that we can live with. This is called an accumulation. We have X on the left and on the right, but we can tolerate that this time because we're adding. If we're adding or multiplying, these are arithmetically associative operations, which means we can reorder them as we please, and the compiler will be very happy to handle that for you.

Finally, we have what compiler people call induction variables, which is to say that as you step through your data, you step through it by a fixed amount on each iteration. A compiler can see quite easily that overall there is a reference to array A on the left and on the right-hand side here; that they will never overlap; that they are at a fixed distance from each other. The compiler will be very happy to take that information and carry on to vectorize your program.

You can experiment with the loop structure of the program if you're having problems, to try and break free of them. Sometimes you can unify the inner and the outer loop for instance. That may make the single remaining loop too complex, but the optimizer will focus on the inner loop, remember. So if you have a lot of action in the outer loop, bringing it into a larger inner loop can be helpful. Or you can reverse the order of the loop nest. The compiler will be inhibited if you're striding through your data with a large gap. And often that's the point of reversing the order of the loop nest; to get yourself to sequential data access.

And something we'll talk about later is memory effects. They can be very significant. So if you have a number of loops accessing the same data in external memory, it may make sense to unify those into one larger loop so that you can fetch the data once. And remember, for vectorization, if you can't get what you want from standard C you can use the explicit fractional built-ins. Here we see a vectorizing built-in, which mandates the compiler to do a dual operation. The compiler doesn't have to figure out if it's safe. You've told it exactly what you want.

When you're tuning up tight loops like this, it's very important to get the compiler to engage the hardware loop. The Blackfin has a zero overhead hardware loop facility, which is very much more efficient than just branching back conditionally to the start of the loop. It also, more subtly, signals that this loop is well understood. If you can get the compiler to emit the LSETUP instruction, then it means that the compiler is very happy with the loop and it'll probably go on to do the vectorizations and the software pipelining. Conversely, if you don't see the hardware loop construct, it may be an indication that the compiler's not entirely happy.

Things that may stop you getting this hardware loop construct would be - well, first of all there are only two, so it's only worth looking for it in the two most deeply nested loops. Secondly, if there are any calls in your loop to functions which puzzle the compiler, then it won't use it. If the function is very simple and local so that the compiler can see exactly what it does, then you may still get the hardware loop. And there must be no transfer of control into the loop, other than the normal entry. You can get away with jumping out of a loop and still get the hardware loop construct, although it lowers the efficiency of the whole loop.

So, let's just go back over the vectorization conditions, because I don't want to make this sound too complicated. There are really a very short list of conditions for successful vectorization that you can tick through on your fingers. You have to have sequential memory access, because we're going to need to fetch two operands at a time. You have to have a known alignment of that data because the Blackfin will only fetch four byte values off four byte boundaries for instance.

Thirdly, the compiler is going to want to be confident about the iteration count for the loop. It's not going to want to vectorize it if the loop doesn't go round very often, and odd numbers of iterations may worry it too, and generally cause it it generate Prolog and Epilog. And fourthly and most importantly, the aliasing of the data. This question of whether two data arrays or buffers referenced in the loop might overlap. If there's any question about that, the compiler will hang back.

To look more carefully at this alignment of the data. You can assume the compiler will lay out the data to assist you, for instance all top-level arrays will be allocated on four-byte boundaries. But you can also help the compiler by passing the addresses - when you pass the address of an array of data, you should pass the address of the start of the array. If, for instance, you have a flag or a count in the zeroth element in the array, it's not helpful to pass the address of the first, second, or third element of the array. That will confuse things. You should stick with the start of the array.

The compiler will attempt to assist you when it's not certain of alignment by planting two forms of the loop. It will plant a decision, say is this data aligned or not? It will evaluate that at run time, and it will jump to either the aligned vectorized loop or the nonaligned, non-vectorized loop, which is terribly helpful and it helps on the performance. But you've made that piece of code twice the size it needed to be. So it's better to be clear with the compiler to begin with.

Here we see an example of data layouts that help and hinder. If for instance you're working with complex numbers, two sixteen bit parts. The real part and the imaginary part. It can be counter-productive to lay out like this with two separate arrays because now the compiler's got to load from two different places, sixteen bits at a time. If you were to combine those, alternating real and imaginary parts in a single array, one 32-bit load fetches both parts of the complex number.

Another thing we see quite often is two dimensional or higher arrays, constructed with great flexibility using Malloc(). Where there is a row of pointers and then each row of data beyond that is a separate Malloc() request. This is very elegant and flexible, from the point of your programming, but it really inhibits the optimizer because the optimizer doesn't know how one row relates to another. It doesn't know that if they're adjacent in memory and it sees no constant offset between the rules. So it's very destructive of performance. Far better to do a simple two-dimensional declaration of the array.

Here's an example of striding through memory that I was mentioning before from a customer benchmark. In the first example we are selecting data elements based on "k", which is the inner loop control, times a constant. That means we're going to stride through the data with a gap of that constant size, which is going to inhibit vectorization. But if we notice that this an associative operation, that we're adding and multiplying so we can reorder these terms, we can get to the

second form of the program where we're now stepping through sequentially controlled by "k". And the program went about six times faster, as I recall.

So just to bring you back to this big problem of aliasing, which tends to be the worst part of getting the compiler to optimize automatically. Pointers coming in from outside can be prime causes of suspicion, arguments, globals. Pointers which have several purposes, and in C++, reference parameters. So basically you should look at all your pointers and you should ask yourself, is it possible for the compiler to know which piece of data I'm referencing at all times? Or do my pointers come across as ambiguous to the compiler?

Perhaps I can make this clearer with the example at the bottom, that if you were to use a global variable to control a FOR loop, then the compiler would worry every time that you wrote to an array in that loop; it would worry that this array might overlap the global. Whereas, if you'd used a local variable, it might be much clearer for the compiler.

Now, we'll see a pattern here as we go through this talk, that there is always a top-level way of doing it with no change to the program, and then varying levels of intervention. In the case of signal processing loops, your tool is the interprocedural analysis. It looks across the whole of an application and it compiles all of that application and gathers information from each of the modules as to what functions are called with what arguments. And what's known about the alignments of global variables and propagation of constants. So this basically passes information back to the modules that have loops that we're attempting to vectorize. Then there's a second compilation to take advantage of that information.

This will never be as strong as the programmer clarifying things to begin with. Attempts to pass information around in the same way a human programmer would. You should remember that this is a context-free analysis. It doesn't know the exact path that you will take through your program. It tries to work out all paths simultaneously dealing with sets of possibilities.

Subchapter 2c: Pragmas

Now, my preferred solution to this kind of problem would be a local, portable intervention like this. We have pragmas here, "noalias" and "vector_for", which can be inserted ahead of a loop. These will assert some condition for the loops to the compiler.

In the case of "noalias", it simply says that none of these loads and stores will overlap. So the compiler no longer has to concern itself about that. Of course if this assertion is untrue, you would get incorrect answers from your program. Another way of doing the same thing is to use the qualifier "restrict" on the declaration which means, on a pointer, that this pointer cannot create an alias.

We have the very similar pragma "vector_for" down here, which tells the compiler it's safe to execute two iterations of the loop in parallel. I would stress these would be regarded as comments if you took your code somewhere else, so they really do no harm. On the question of alignment, it's possible to assert to the compiler what the alignment of the data is, which can be very useful. Unfortunately this one isn't a pragma. It's an executable statement. So you have some restrictions on where you can place it.

But typically, you would place it at the start of a function to inform the compiler of the alignment of the data elements that have come in as parameters from outside. We have the pragma "all_aligned" that you can place in front of a loop to tell the compiler that everything is fine with the alignment. If you do get yourself with odd aligned data coming in, you can even put an optional parameter on this to tell it that the data is all misaligned, so that it will generate a prolog to the loop and then carry on from the first aligned position.

The final pragma on this slide is pragma “different_ banks” which is useful in encouraging the compiler to do two loads or a load and a store simultaneously in the same cycle. That can cause a sub-block conflict in the memory. And this pragma encourages the compiler not to worry about that. The compiler doesn’t know a lot about the layout of the data in memory. So this is a programmer saying that they’re taking care of the data layout and there won’t be a problem.

If you can code your FOR loop with constants or literals then that’s ideal. If you have to use variables, then you can use this pragma loop count to give the compiler some information about the behavior of this loop. The first parameter is a minimum trip count to say that the loop will always iterate at least this many times. The middle one is the maximum trip count, which tells the compiler the maximum times the loop may iterate and that can be helpful to the compiler and seeing it’s going to be worth making a big effort with this loop. And finally, the trip modulo which is used during software pipelining and vectorization to help the compiler see how you stride through the data.

Subchapter 2d: Volatile etc

Another very useful qualifier is Volatile. Volatile tells the compiler that the data may change due to some external agency. If you think about it, this is always going to be a possibility, any real application is going to have data coming in from outside by DMA transfer or from some sort of hardware device. And it’s very important the compiler knows that it can make no assumptions about this data that may be changing underneath it. In fact, missing off this Volatile qualifier is now the largest single cause of the Support requests because the compiler optimizer has become so aggressive and so powerful that it locates loops appear to be doing nothing and eliminates them. So it’s very important to put that Volatile marker on, to let the compiler know where there’s an external agency. And the opposite qualification can also help to mark arrays “const”, if they’re absolutely unchanging.

One of the signal processing accelerators is circular addressing where we run round a buffer in a modulo fashion. This can now be accessed from portable C because the compiler will recognize the use of a modulo operator in an addressing expression like this one at the top here. Now if the compiler is not absolutely certain of the safety of applying circular addressing, but you are, as the programmer, you can use the option force-circbuf to the compiler to reassure it that it’s safe to make this optimization.

Finally, we have two explicit built-ins to absolutely mandate that the optimization is done by the programmer.

One thing to bear in mind is that there are parts of this that the compilers still don’t reach. For instance, the automatic bit reverse addressing that you find in an FFT is not going to be generated from portable C by a compiler. And anyplace you have a single really complex instruction, such as the [powerful Viterbi instruction that you have on the Blackfin, or a Bitmux.

We had a customer benchmark a few weeks ago where the insertion of a bit MUX instruction, made it go 25 times faster in the critical loop. And this is not unusual. It’s very important to know your vocabulary on these machines, to be familiar with the built-ins that access these individual, very powerful instructions and to be able to insert them in your program. Here’s another example of this vocabulary. The “Count Ones” intrinsic. This is a real customer example where in portable C you’re counting the occurrence of bits in a word, but if you draw on the correct intrinsic, then it goes 60 times faster.

Finally, we come to the ultimate deterrent. If all else fails, you can program in Assembly from within your C program using these inline asm() statements. They’re modeled after the [GNU] asm statements although they’re not exactly the same. These used to be quite a problem

because the compiler optimizer would become uncertain of everything after it had seen some assembly statements in a loop. But they have a registers “Clobbered” field now by which the programmer can tell the compiler exactly which registers are affected by the asm() statement. And that makes the optimization flow round them much better and they have become much more useful.

One area of ambiguity remaining is that if you modify data using asm() statements, then you assert in the clobber field that memory has been amended. But this obviously a very vague assertion: imprecise. So on the right, we see an example of how you would use these asm() statements, in this case we’re generating an arithmetic primitive that’s useful in signal processing, which isn’t present on the machine or in the language in any other form. So you can construct your own arithmetic primitives in this way if they’re not already present.

You can also put these register clobber strings onto C functions if you find it helpful- for instance if you’re calling a little assembly function to let the compiler know what that function’s going to be modifying.

Okay, that concludes the section on tight arithmetic loops.

Chapter 3: Tuning Control code

Subchapter 3a: Conditionals

And now we’re going to be turning to what we call control code, which really means more normal portable C where you have more branching around going on, more decisions being taken. As always, there’s a tick box that if you use the profile guided optimization, then the compiler will automatically try and solve most of the problems for you.

The way that this works is that your program has to be simulated. And during the simulation a complete trace is made of every transfer of control. I would recommend using the compiled simulator for this because it’s hundreds of times faster than the cycle-accurate simulator. So now the compiler knows which way every branch went, so it’s comparatively simple for the compiler to take a second bite at this. Recompile the program using this new knowledge and have every branch as optimal as possible. It won’t help you if your branches have a 50-50 probability of which way they go. But if they predominantly go one direction or the other, then you’ll see an improvement. 10 to 15% improvement on an application is not unusual.

You should bear in mind that this information is derived from whatever data you run the program with, so you should choose appropriate training data. Sometimes people are more concerned with the worst case than the average case, for instance. Something can be a little disturbing about this, is that the compiler is very aggressive. It doesn’t go for statically predicted jumps, it looks for the drop through. If you have a conditional expression on this machine, and you don’t jump away, you just drop through to the next instruction in sequence, there are zero stalls involved. So that’s the pattern the compiler aims for. But that pattern means it may have to substantially rearrange your code, so suddenly your blocks of assembly have moved around on you after this optimization.

Or, if you don’t like the manual option or you can’t use simulation; if you like the automatic option, you can do it manually. You can insert “expected true” and “expected false” into your C so that now you’re advising the compiler on which way you expect branches to go. For instance, here, we see it used with an error call. You check to see if you should call the error function, and very unusually, you would enter the loop and call the error function. Well the presence of “expected false” tells the compiler that this is a very unusual thing to happen, and it will rearrange the code

so that you branch away to the end to call your error function, or more commonly, just drop through with zero stalls to the next instruction.

Here we see a simple substitution of what looks like a conditional expression, conditional branch, which can be replaced by a “max” operator. We have mini, max, and absolute available on the Blackfin as individual instructions. So we remove a conditional branch, which might have taken many cycles and replace it with an always single cycle machine instruction. A compiler will do this for you in the 32-bit case and some 16-bit cases, but occasionally you have to help it with the 16 bits. And you should note that min and max are for signed values only. So it’s another case where you really need to think about whether your data needs to be unsigned.

Here’s another way of removing these conditional expressions. If you have a loop with “if” in it, or “if then else”, there’s always going to be some conditional branches in there with a large number of stalls. But you could recode it like this. You could invert the pattern. You can have the “if” on the outside and duplicate FOR loops in the two blocks, and that might run very much faster at some expense in code size.

The Blackfin has a very fast conditional instruction here, called a predicated instruction, where in a single cycle it tests a condition and then does a conditional register assignment. Just one cycle.

And that’s what you want to see in the codes the compiler generates. If you don’t see it, it’s worth considering whether you can shuffle the code around a little to encourage the compiler to generate this form. To think about what might have stopped it happening. Typically, you find yourself considering what we call speculative execution. For instance, if the original expression had been if a<A>, then X is <expression one> otherwise <expression two>. If we were to convert this to one of these forms down here, where we execute <expression one> and we execute <expression two> unconditionally in every case, and then get to the conditional bit and we do an assignment. This last one is much closer to the pattern of the machine instruction, make for an easier lock (ie. Recognition of the instruction pattern by the compiler). But probably the key thing was that we moved these two expressions forward and made them unconditional.

Now what if one of those expressions had been something that might cause an address error for instance? That would mean that the compiler would say to itself, I can’t move that, that might cause a program error. So this is a programmer level decision. It’s good to get clear on what a compiler will do for you and what you ought to do as a programmer. Another area in which speculation occurs is when you software pipeline a loop, you often find yourself wanting to load one extra data element off the end of the array. And then we find programmers wanting to place their data right at the end of the virtual memory, so that would be an illegal access.

So the compiler plays it safe and won’t do that software pipeline by default. But we give you a compiler option -extra-loads where you can tell the compiler it’s safe to speculatively load off the end of the array.

Here’s another example of removing conditional branches. This one’s really a trick, but I have it here just to, as an encouragement to think creatively about these things. It’s a real customer benchmark. The code comes in and checks a flag array, perhaps a Boolean array to see if the condition’s true or not. And then it adds buffer times 64 or subtracts buffer times 64, depending on that flag.

So, what if instead of having one or zero in this “key” array, we had plus or minus 64? Then you can have a loop expression, which has no conditionals at all. Very much faster. But this a good example of something a compiler won’t do for you. The compiler is not going to change the contents of your data arrays for you. This is a programmer level decision.

Subchapter 3b: Division
Now, like floating point, there is no support for division on the Blackfin as a single cycle instruction. There are instructions to handle one bit of a division, but the total cost of a 32-bit division is still going to be significant. So you should get division out of loops wherever possible. And you should bear in mind that the modulus operator also implies division. The classic trick with this is the power of two. If you're dividing by a power of two, then you can convert that into a right shift.

In the case of an unsigned divisor, that would just be a single cycle. But to be absolutely safe, if the divisor is signed, the compiler actually has to do this tricky little expression here. And it takes the six cycles of machine instructions shown to do the safe shift for signed variables. So that's another case where it's worth thinking about whether you really need to be using signed or unsigned variables at that point.

And remember that to even consider this; the compiler must have visibility as to the nature of the divisor. If the divisor is a constant or a literal fine. The compiler can see if it's a power of two. But what if it's a variable or a global? Is there any way that you can expect the compiler to know what's in that variable? Or should you simplify the program?

The division support will split into two, depending on the precision that's required. Wishfully speaking, if the result and the deviser fit into 16 bits you'll get by in about 40 cycles or less. But if we need a full 32 bit division, it could be as bad as 400 cycles. So that's very significant and well worth saving. So it may be worth your while thinking about scaling values before you come into this division, so that you get the 16-bit case.

And something unexpected is that the compiler may generate a division if it's confident that there's an inner loop that's going to do a lot of work, and it wants to use the hardware loop construct. That construct needs to be fed the number of iterations of the loop. And if the compiler is dealing with variables, it might think it's worth risking a divide to establish the number of iterations from the FOR loop variable. So, it would have been far better to use constants or Literals to begin with.

Here's another interesting little trick which illustrates another of these programmer / compiler divides. In a real customer benchmark, we have a testing of ratios. If x divided by y is greater than a divided by b was the original code. Now this can be made to go hundreds of times faster by using high school algebra to substitute the expression so it becomes if X times B is greater than A times Y, and we have lost those divisions we've got simple multiplication. But the problem is, this only works if we avoid overflow. If x and b, for instance, are 12-bit values, then fine. The result of multiplying them is 24 bits which will fit in our 32 bits. But what if it was 17 bits each? Then we have a potential overflow. So the compiler is not going to risk overflowing your data. It's only the human programmer that thinks about precision in between eight, 16, 32, 64 bits. So the programmer has to make this decision. It's not an automatic compiler optimization.

Subchapter 3c: Advanced
One of the ways in which the Blackfin design saves time and power is by associating registers in groups with functional units. For instance, you have arithmetic registers or d-regs and you have addressing registers or p-regs. But it costs stalls to transfer values between these. It won't happen often, but when it does, you might want to think about saving those stalls.

The way this can happen is for instance with complex address arithmetic. If you're stepping through an array sequentially or with a two or a four gap, then that's fine. But if, for instance, it was a structured array with an odd sized element so you're trying to step through by nine or eleven or something like that, then the arithmetic may be taken over to the d-reg unit where we can do arbitrary multiplication and then brought back. So you have two sets of stalls transferring that calculation back and forward. So that's another reason to think about simplifying your

address arithmetic and being very aware of the repertoire of address arithmetic that happens quickly.

The Blackfin has these two aspects, the DSP and the microcontroller. And one of the places where they join is in how big is an integer? For DSP style calculations you'll usually be working with 16-bit data. But for microcontroller, decision making code, you'll be working with 32 bits. You have to remember that the instructions set architecture is not symmetric for all these values. In other words it doesn't support all operations for all sizes.

And when we come to something like a conditional branch, we're doing comparisons in 32 bits. So, if you declare a "short" as a local variable, and then use it to do a comparison, the compiler is going to have to promote that up to 32 bits every time. So better to start off with 32 bits for your control code variables and leave the 16-bit stuff for the data arrays. It is especially true of loop control values. The compilers always worry with 16 bits about whether they're going to get an overflow.

Another aspect of the integer size is the multiplication cost. A 32-bit multiplication will take you three cycles, whereas with a 16-bit multiplication its not only single cycle, you can even vectorize and do two of them in a single cycle. So it's six times faster. So you want to be careful that you do as many 16-bit multiplications as you can rather than 32. And one place this can sneak in is, as discussed previously, the address arithmetic. If we're having to multiply up a strange address increment, it will get done at 32 bits as far as Standard C.

The average control code is typically written with lots of fairly small functions. And these inhibit optimization. If the compiler doesn't know what they might be getting up to, what they might be changing. So we have the opportunity in the compiler for an automatic optimization to just tick the box and ask for function inlining. This can be a big saving because inlining a function, which means taking the body of that function's code and inserting it at the point of the call, it will save you the call instruction which was multi-cycle. It will save you the return, which was multi-cycle, and the construction of parameters and return values.

But, just as important, it gives the optimizer greater visibility. The optimizer can see exactly what that function was getting up to now. And, also, we have increased the size of the piece of code between transfers of control. The call to the function was a transfer of control, removing it leaves you with a larger piece of code to work with. And having a larger piece of code is very important to the optimizer because it gives it more scope to move instructions around, to schedule them into any stalls and increase the efficiency.

Of course, the flip side of inlining functions is that you dramatically increase code size if you're not careful. So automatically you tick the box, manually you would use the inline qualifier to control which functions get inlined.

This is just a minor aspect of control code programming because we have this hardware loop construct, we should be aware that it's not an exact match for the C definition of a FOR loop.

A FOR loop in C may iterate no times at all, whereas a hardware loop assumes it will iterate at least once. And the result of this unfortunate mismatch is that we have to plant guard code from the compiler in front of loops, checking to see if the loop's going to iterate at all. This is a little tedious and it can easily be removed by using constant bounds or the pragma loop_count that I mentioned earlier.

Chapter 4: Memory Performance

Subchapter 4a: Memory costs

Okay. So now we're going to talk a little bit about memory performance, which is perhaps surprising for a C optimization talk. But the hard truth is that as processors have got so much faster, memories have not been increasing in speed, anything like the same pace. And there's an enormous performance cliff now between external memories and the onboard L1 memories. And you simply have to be aware of this. It's become part of the skillset if you want to speed up a C program.

When you're setting up your Blackfin, you will normally choose what power you want, and hence, what speed, or you'll decide you want to go as fast as possible and choose your speed. But if you find you've got a memory-bound application where it's the memory access that's your biggest cost, then I consult a little table like this, which shows me the bus speeds that are available to me, because if I ramp up the bus speed a little, I can improve the performance of the whole application. I may not get precisely the core speed I originally had. So this is for use when the memory bus is the important thing.

The yellow bars show combinations that are available, that are very useful from amongst those that are simply theoretical. This granularity occurs because on the EZkits we're driving with particular clock speeds. If you have a variable signal, clock signal generator, then I guess you can adjust this infinitely. So, the automatic tool for handling memory problems is caching. That we just switch that on and let the device load into L1 memory the data that's being used and hold it there as long as it's being reused.

This is switched on quite simply with a linker option and a control variable in your program. These bits are controlling individual instruction and data caches, but you'll find that discussed in other talks. There's also an interesting thing called "write back" mode, which tells the machine that it doesn't need to keep copying back data from the cache to the external memory. And that requires a change to the CPLB table to activate, you'll see later.

So, apart from the caching, the other way of simply speeding things up is to load really critical pieces into L1 data memory explicitly. And the way you would do that from a C program would be something like this. You'd put in a "section" directive. This name "L1data" would be found in the data layout file, the LDF file. Memory costs come in several flavors. As well as the cost to access the external memory, there's also the cost of switching the rows in external memory. Every four kilobytes in a typical blackfin setup you have a different row. And opening a row and moving to the next one adds memory cycles. You can avoid these by spreading your data across the different banks, and we'll show the effect in a table in the next slide.

What I want you to know straight off is there's an easy way of getting a rough solution to this, which is to use this macro "PARTITION_EZKIT_SDRAM" with the standard LDF file, which will put your code in one bank and your heap in the next and your data in the next and so on. So simply coding that phrase on the EEMBC Mpeg4 encoder which is a large typical program, got a six and a half percent performance improvement, without any deep understanding of the memory issues involved.

If you want to go a little deeper, here's an interesting table. And the reason I introduce this to a talk on C is because you really want to appreciate the size of these numbers. If you're working with L1 memory, you're in this top row. Everything costs one, that's just fine. But when your program spills out, say to external memory, then your costs are in this third row (L3). Reading sequentially 16 bits, going to take you 40 cycles. If it's not in the same row and you have to open and close rows, 141. I mean, these are huge amounts.

Of course you'll rarely see them if you program correctly. For instance, if you have the cache switched on, then you're in this second row. And you're seeing a much smaller cost for reading

sequentially, even if you alternate the rows, it's still not too bad. And if you spread across the banks, you avoid the alternating row cost entirely.

But 7.7 is still not one. That's because what we're doing here is reading sequentially. That will be like streaming data into a signal processing application. Caches are really set up to give you the maximum advantage if you reuse data. So you want to think about the pattern of your data use. Get it into the cache and use it repeatedly, rather than fetch it from external memory over and over.

DMA might be the answer, as will be discussed in other talks. Similar situation with writing single cycle costs if you're in L1, if you're cached, the numbers rise. This number here with alternating rows on our right, would be the place where you would consider applying your write back mode. I would stress this is not a problem with the Blackfin. This is arises entirely from having a high core speed and memory speeds that top out much lower.

Subchapter 4b: Code Speed vs Space.
The other interesting aspect for the C programmer in considering memory, is to reduce the space the code occupies because you're going to try to fit this code into an instruction cache or into L1 memory. And if it's smaller, more of it will fit into that fast memory. Also, if you have a very great deal of code, it's going to sit a lot of time out in external memory, and that costs money in your final deployment. So the compiler tools offer you a choice of how to optimize your program. By default we optimize it for maximum speed, but you can also request minimum code size. What that means is the compiler will avoid any optimization that might expand the code.

This situation is at its most extreme in the DSP kernels where we like to spread the loop out aggressively. So what you can do is you can give guidance to the compiler down to quite a low level. You can not only decide one file should be compiled for space, another for speed, you can come right down to the function level and put in these pragma markers we see at the bottom to control the process quite finely.

When you look at what's happening, when you compile for speed and for space, you can break down what's causing the changes. You can see as much as a 50% variation in code size by working these options. And about half of the difference is caused by the loop expansions and the optimizations that grow the code. Another half is caused by function inlining. Which means it's very important to get a grip on which functions you want to be inlined. And then there are some minor stuff.

Interesting one here is the -Ofp option which balances the stack and parameters to maximize the use of 16-bit offset, and thus reduce the code size. Every little helps. I would emphasize that the space optimized code has no restrictions on functionality at all. It can do everything that the speed optimized code can do, including interrupts and so on.

So, as usual, there is an automatic optimization to try and get you this minimum effort. There is an -Ov slider which allows you choose a value from one to a hundred to indicate your preference between compiled mostly for speed or mostly for space across the breadth of the application. This is based on the profile guided optimization. Remember that was when we ran the program in a simulator and we now know exactly where it spends its time. So the compiler has a good idea now of how important each block of code is to performance.

So the blocks of code which are infrequently executed could be compiled to save space rather than to maximize speed. The compiler combines this knowledge with its own understanding of exactly how space expansive each of its optimizations are, to give you a very flexible, tailored solution. This is very simple to the programmer, and very effective. Quite complicated in the compiler.

If we graph the kind of results we get, we get interesting shapes like this, that show that at the extremes of minimizing space or maximizing speed, we are getting very low returns. But we have a nice sweet spot here near the origin where several points are clustered. These are corresponding to the values 30 through 70 roughly on the Ov slider. And any of those would do to give us a good tradeoff between speed and space.

If you're not convinced by a fully automatic optimization, or you can't use the simulator, you can approach this kind of problem manually. Here I've drawn up a table of what happens when you take each of the files comprising this application individually and compile them for speed, compile them for space, and then measure how much effect that had. It's very interesting. In this column, compiling for speed, only the yellow files have a significant effect. Which tells us most of our application could simply be compiled to save space. It's got no great effect on speed at all.

So immediately we throw twenty out of twenty-eight files into a bin and say that's stable. Let's leave that alone and just look at the remaining files. The right-hand columns show the effect of compiling for space. And I highlighted in blue those files where there is a significant effect. And interestingly, we see they don't always match the yellow files, which means we could compile up some of these to save space at no cost to performance.

So, from these tables, you can extract yourself a prioritized list of tradeoffs. So you can run down this to make a few decisions to tailor your application. So, I offer this an alternative to relying on the fully automatic optimization, to break down the problem and you'll be surprised that you don't have to worry about every single file in an application.

Chapter 5: Examples

Subchapter 5a: Data Structure

Okay, the final section is just a couple of examples based on real user benchmarks that we have sent in to try and go back to some of the things we've discussed and show you them used in context.

First example's very simple. It's of a complex data structure. We get a program in here which is obviously doing a couple of MACs. So we're glance at the code and we don't see vectorization; we see a single MAC on each line.

So, you might be satisfied with this as a programmer. You might already have your performance target, and you don't want to go deeper, and have to stare at a lot of code. Fine. But if you need those extra cycles, you need to worry about why didn't you get that vectorization? So the reason would seem to be that we're doing three loads, and we can't do three loads in a single cycle. So if we think back to what we said about complex numbers, we have the 16-bit imaginary part and a 16-bit real part adjacent. So why can't we pick them up in one 32-bit load?

Well the reason is because they're part of a structure. There's another 16-bit element in here. So we have a six byte size. So that means that in one element of this struct array, this will be an aligned address, and the next one it's going to be a misaligned address, so we can't do our 32-bit load. So, the compiler's not going to fix that for you, but as a programmer, you can consider "Can I change this data structure?" It turned out to be well worthwhile, simply making a copy of the data into a local array with the real and imaginary parts adjacent and then driving off that.

So we didn't so much change the user's data structure, as make a local copy of relevant data, and still there was a big payoff. We got the code structure we wanted. The optimal vectorized MAC. A little thing to notice here is that there's a number of possible ways you could express this,

but using this [K*2] is a particularly strong way of hinting to the compiler that the alignment will be four bytes in this short array.

Subchapter 5b: Whole Application

And now the final example, which is a complete evaluation from start to finish. We were given this whole application to see if it would reach the customer's target on a Blackfin. Do a quick evaluation of that. So the first thing we always do is insert the timing points and checking code so we know if we have perturbed the application. At that point we're running at fifty million cycles. Now, you switch the optimization on, you tick the box, and you're down to a nine and a half million cycles, which reinforces why you really want to do that.

And then we switch the cache on, which took care of a lot of nasty memory effects and we're down to 1.4 million cycles. So you got a very fast running start. Next we would investigate global optimizations; the IPA the function in aligning and so on to see how much effect they would have, and then on to the more interesting stuff. We would do the statistical profile and get ourselves a list of hot spots which we would go through in priority order to see if any of them could be improved; to see if we understood the situation. Finally, we would set the exact bus speed in MHz and do any memory tuning.

So let's see how some of those hot spots went.

The first hotspot was kind of embarrassing. We have this nice implementation of ETSI fractional arithmetic, which the customer's program was using by default. But it seemed to be going slow, and when I looked at the code, I saw calls to ETSI functions rather than single machine instructions. And the reason is that the customer didn't know that we have this flag, #define ETSI_SOURCE that you need to set to put you into ETSI mode.

So we improved the documentation on that. You set this and the code improved dramatically, and this section from a 187,000 to 141,000 cycles.

The next hotspot was a critical loop which had three conditional jumps in it. It's using 16-bit data and it's got an "if" here and it's got an "if" statement there, and an other "if". So when we looked at the code, most of the cycles were stall cycles on these conditional branches. The total initial cost 67,000 cycles.

So, let's think about whether we can get rid of some of these. The first conditional jump went out to using a "max" instruction, which is in the standard C library, so you can express it simply that way or it can use a builtin. And that cut the time 67,000 to 40,000 cycles. The reason the compiler didn't do it for you was because you were dealing with 16-bit data.

Now, let's look at the next conditional jump. It's of this form. If we satisfy a condition, then store it in memory. So what I did was use MAX again. Now we're unconditionally storing to memory, and the reason the compiler won't do this for you is because it's changed the memory traffic. It's writing out every time now. That not only puts more load on your memory bus, but there's a possibility of an address error on this expression. So that's definitely a programmer level decision. That saves another six and a half thousand cycles.

The third and final conditional jump looked like it should translate as a predicated instruction, a single cycle condition. If CC then register assignment. So I rearranged the code to get it into something more like that pattern, and the problem was that we're now doing more stuff unconditionally that used to be conditional. The compiler is not going to risk that for you, you have to force it.

Now every time we're going through doing this store. So, we saved another sixteen and a half thousand cycles there.

Something we did along the way was change the definition of this variable "maxval" from two bytes to four bytes, which saved a coercion and saved us a cycle, which was worth 2,000 cycles over the application.

So here's what the final code looked like. Not a conditional jump to be seen. Very short code. In fact, this could go one cycle faster if software pipelining was invoked and we discovered that could be done by using the pragma no_alias.

So the loops now running three and a half times faster than the original, with no jump stalls at all. Now, I would stress that you don't have to do this. These kind of techniques are available if you are desperate for those cycles. You don't have to mess up your program unless you want to save a lot of time.

The next focal point from the statistical profiler was memcpy(). The customer was moving data between buffers in external memory. That was quite difficult for us to intervene in, because remember this is a customer application, so we don't understand everything it's doing. So we experimented with simply invoking the "write back" cache mode. And got a very significant saving in time.

You invoke the "write back" cache mode by changing an entry in a table file, CPLBtab.H. And you see how it's done on this slide.

Something you should know about memcpy() type functions is that they are written to reward four byte alignment of data. This is a feature of the Blackfin memcpy(). It will dynamically check whether the data coming in is aligned and switch to a word level loop if it is. Otherwise it does the traditional portable C byte by byte copy. And the point is that the byte by byte copy is very much slower. In fact the 32-bit word loop is eight times faster than a byte-by-byte copy. So it's worth thinking about whether what you're feeding into the memcpy() is correctly aligned. Also, it may be counter-productive to feed requests to move one or two bytes at a time into the memcpy().

So we talked to the customer about how they might possibly realign some of these application buffers to exploit this feature maximally in the future.

The next point thrown up by the statistical profiler was this loop: where we see four MAC operations and when we looked at the code, we were only using a single one of the two Blackfin accumulators. And experimented with the code to see what the problem might be. Well, it may simply be the complexity of what's going on. So what we did was we split the loop, split this loop into two single loops, so that it looks like this now. And we have two MACs to match our two accumulators in each of the loops. And that worked fine. We got out this kind of code, which is optimal, vectorized dual MAC. And 20,000 cycles were saved.

We're getting to the end now. The final step in moving to finishing an evaluation is to get onto the target platform, which was a BF532 this time. We've been working on a BF533 up to now. We've come quite a long way with what's been discussed. We've come from one and a half million to .73 million cycles. In other words, we've doubled the speed of the application by going through these hotspots that were identified by the statistical profiler.

And setting the bus speed got another useful gain. And I see, here we are, setting the bus speed up. This application was memory bound so we maximize on the bus speed, setting it to 130.5 MHz.

And finally, we consider what to do with the L1 memory. The default situation is that the tools will put whatever comes first into your very fast L1 memory. So if you're a bit more selective and you put data that's used most often into that L1 memory, you can gain speed. The trick is to know, if you're not the customer, what data should go in there.

What we did was we used the hint at start. We studied the statistical profiler at the mixed instruction view level and we found a store instructions that was taking twelve times longer than anything adjacent indicating it had a memory problem. So we tracked that back to which user buffer it is and put that buffer into the L-1 memory. And then we checked the cache behavior. There's a cache viewer with the tools, and you can see if there is any piece of data causing a lot of conflicts in the cache and move that to the L1 data. So, this final speed up go as to 747,000 cycles. So that was a successful evaluation and the customer was well pleased.

But the point I'd like to end on is it's all optional. If you port your application in and it's going fast enough, fine. You only go into this stuff if you want that extra bit speed. You'll find reference material on the Blackfin site. The engineer note with the latencies that I discussed earlier, and you'll find a very useful chapter in the Blackfin compiler manual, "achieving optimum performance from C". Which goes through a lot of these techniques in more detail.

And if there are any questions, we're always glad to hear from you. We monitor the Blackfin support mill for people who have performance problems. Thank you for listening to this talk today. I hope it's been useful.