The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

**PROFESSOR:** All right, everyone. Welcome back. Today we're going to talk about the profiling tools available on Cell to help you evaluate how your programs are running. In the next hour I'll talk about SIMD, which is how you can take advantage of the hardware support for data parallelization.

First up, I'd like to follow up with some questions that we had last time about gdb, and then I'll discuss some of the various methods for profiling on Cell. I'll talk about the Cell simulator and using the decrementers, and then we're going to talk about static profiling and instruction reordering, which I started to get into last time.

So first up, we ran into some problems last time with gdb, various error messages. First thing, sometimes gbd is going to examine the wrong variable, as we saw when the names are ambiguous. One thing you can do to get around this-- we had problems with ambiguity between variables, which have the same name where one of them was in the spu thread and one of them was in the ppu thread. And so to work around this, one thing you can do is use spu-gdb, which will just log on to a particular spu thread, then you can just look for variable names in that thread. You can also, of course, rename your variables.

Second problem is that sometimes gdb will delete your breakpoints before you want it to delete them. Reason being, it will remove breakpoints whenever the program associated with a breakpoint has been unloaded. But sometimes you have more than one thread running the same program, and so the second one you won't be able to break into.

So again, you can use spu-gdb to log on to a particular thread if you want to do that. Another thing you can do is for example put a delay loop at the end of your spu

thread in order to keep that thread running for as long as possible. It seems that as long as gdb doesn't unload that thread, then it won't delete your breakpoint.

And finally, I saw this kind of cryptic error thread event breakpoint gdb should not stop. Indeed it shouldn't. One thing, to get around this, it seems that this problem doesn't happen on spu-gdb.

So I'll just recap how to use spu-gdb to debug your programs. Last time we were using mostly ppu-gdb, which will help you debug kind of both your PPU program and your SPU program together. And you're able to switch between all the threads that are available. On the other hand, when you're using spu-gdb, it's just going to log on to a particular SPU thread. So what you do is you invoke your original people PPU program with SPU debug star equals 1. And then you can background it.

Now what happens is every time that your-- every time that your SPU-- sorry, every time that your program spawns a new SPU thread, it will print the process ID and then actually that thread, the spawned thread, will stop and wait for a debugger to attach to it. So at this point, you can run spu-gdb -p and then the process number that they give you. And when you attach to it, you can continue that thread to resume debugging. Any questions?

OK, so the Cell simulator is one of the tools that we have available to help you debug. This is how you invoke it if you have it installed, opt/ibm/systemsim-cell/bin/systemsim, and you want to use -g to use the graphical interface. And a little window will pop up and you'll click Go. The window looks like this. It actually, from here, you can have a lot of fine-grained control over how your program is running. In fact you can advance the state of the simulation cycle by cycle to look at what's going on inside the simulated Cell system.

And when you boot up this simulator, you can actually get a complete Linux environment in there with an x term. And what you can do in this Linux environment is you can copy over programs and then you can-- you can copy over programs which were compiled for the Cell, and then you can run them. In order to transfer a program to your simulated system so that you can run it, you can use this function--

or you can use this program call through which is available on the simulated system.

And what it will do is it will grab onto a file which is on the host system and copy it in. And it will just put it on standard out so you can redirect it to whatever file you want. And then when you make it executable, then you can run it. All right, questions?

While you're in the simulator, you can view a lot of different things. You can get a lot of information about the state of the machine. In particular, you can look at the registers at any particular time, again with clock step by clock step. This might not be as interesting now that you have the debugger. And you can run the debugger on real hardware.

But one thing which the Cell simulator does get you is these dynamic profiling tools. And what it will do is it will look at the state of the simulation and it will figure out where things are stalling, where the time is being spent by the processor. And you can get separate stats like these. Let's see, it tells you, for example, total cycle count stalls for various reasons. You get all these statistics separately for the PPU and each SPU. So this can be very helpful if you're trying to do, for example, you're trying to figure out exactly what's going on each SPU, do load balancing, whatever.

And so these statistics will hold-- these statistics will cover whatever length of the program you run. But you can also get more fine-grain statistics using these profiling functions. So if you include profile dot H in your program, then you get access to these functions, profile clear, start, and stop. And what these will do is they actually-- when they get compiled into your program, there are no ops when you run them on a real Cell processor. But the simulator will be able to latch onto them and you can use those to start and stop regions of interest for profiling.

Now the simulator comes with the SDK, and it's a little bit cumbersome to install. So we haven't made it available yet on the actual PS3 hardware. You can run it on x86, and so if your group would like to get set up with that on one of your computers, or we'll try and make it available on our hardware if there's enough interest. So please let us know.

OK, the next thing we'll cover is using decrementers to profile your program. This is one way you can use dynamic profiling, which actually doesn't require the simulator. So when you get information from the decrementers, you can run these programs on your actual Cell hardware. Basically the decrementer is just a counter that counts down at a constant rate. So you can use it as a clock to figure out how long different events are lasting.

And the rate at which the decrementer counts down is not that fast. So you're not going to be able to use it to time things on the order of a few clock cycles. It's best for timing things which are maybe on the order of thousands of instructions in length. And how you use it is there are these SPU write channel and SPU read channel functions which give you access to some of the internals of the Cell processor.

And so here's an example setup you can use. Basically, first you call write channel, and you pass in this MFC_DECREMENTER_EVENT thing. And what that will do is initialize this decrementer counter for you. And then you read the value before and after the function you want to profile, and when you subtract, that just gives you the length or the time that was elapsed in some arbitrary units.

And of course, it's counting down, so you want to subtract the start from the end. And after you're done using it, you can do another write channel to stop the counter from continuing. All right, any questions?

OK, let's continue talking about instruction reordering, which I started last time. So I'm going to kind of start from the beginning this time, and maybe it'll be more coherent. So on the Cell architecture, when you're looking at what instructions are being run, the instructions are mostly going to be of the form evaluate some function of some of the registers and write the result to some other register.

And the assembly file are what you get when you run with GCC-S is just going to be a human-readable representation of these instructions. And I'll show you how to read that actually later. And you can think of these instructions just as executing in series just in the order in which they appear in the assembly. So if you think of one

instruction starting and then finishing, the next instruction comes in, finishes, that should be consistent with the actual behavior of the hardware.

Now for real hardware, waiting for one instruction to finish before the next one starts is going to be too slow. So there are these various optimizations we pull or that they do on the hardware in order to run instructions sooner than when the previous instruction finishes. So the big one of these is pipelining, where you have multiple stages inside your processor, and you can run each instruction-- you can push each instruction in before the previous one has completed.

And in pipelining, you're going to be subject to these dependencies between instructions. For example, if one instruction reads from a value that the previous instruction wrote, then clearly you can't start the second instruction until after the first instruction has completed. The Cell processor also has-- actually has multiple pipelines in which you can insert instructions and another optimization that they do is branch prediction in order to reduce stalls from branching.

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Pardon?

**AUDIENCE:** Can you talk about static branch prediction?

**PROFESSOR:** Yes, so static branch prediction means-- or, rather the way it's done on the Cell is that what's predicted for the branch is not going to be affected by the history of which branches have been taken. And some other processors actually do use this information. All right, now the thing about pipelining is that because we still have to honor these dependencies between instructions, the order in which we evaluate instructions, that's going to make a difference on how long the program runs.

So let's take the simple example where we have three instructions, a, b, and c, and we have this pipeline processor where we can insert instructions at these clock ticks. So suppose that c depends on b, and there are no other dependencies in the system. Then this sequence of instructions is going to take five cycles, because we can push b in after the first cycle, but we can't push c in until two cycles after that. Is

that clear? Questions?

All right, now because c depends on b, it kind of makes sense to push b as far up as we can in order to ease this dependency. So if we execute b before a instead of after a-- and remember we can reorder b and a however we want because there's no dependencies between them-- then we can start b at the first tick, then we put in a right after that, and then c. And now the sequence only takes four clock cycles. All right?

So observe that, when we put in c in this third clock cycle, we actually couldn't have put in the third instruction any sooner, no matter what the dependencies were. And so in general, we're going to want that to be the case where we want instructions to be waiting on the pipeline rather than on dependencies. So that means in the first picture there's this stall of one clock cycle, and we're going to want to eliminate these kinds of stalls. All right?

So what we're going to do is we're going to use these static profiling tools to figure out where stalls happen. Now the first thing we're going to do-- the first thing that we have to do is generate the assembly and the instruction schedule that goes with it. And so you can use GCC -S, and the same flag works with xlc to generate the assembly. And then we have this utility called spu_timing, which runs on the Cell, and which will tell you the instruction schedule.

Basically it'll take in your assembly file and tell you exactly when each instruction in that assembly gets scheduled. If you call it with -running-count, then it'll tell you the running count of clock cycles at each stage, which means if you look at the last number, that will tell you how long your program took to run in clock cycles. And we'll write the output to a file with the same name as the input but with dot timing at the end.

Now if you're using our Makefile, you can actually do all this, generate the assembly and the timing info, all in one step by doing SPU_TIMING=1 and then make the name of the .s file, all right?

Now if you'll recall on Cell we're going to have only in-order execution. The instructions are just going to execute in the order that they're specified in the assembly. And because the Cell has dual pipelines, that means there's two pipelines in which instructions can go, and the pipelines are going to be selected just based on the instruction type.

And of course, two instructions are only going to go in simultaneously when the dependencies allow it.

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Right. So if you're looking at the assembly, you're going to see lots of lines of this form OP DESTINATION and then the source registers. So when you're trying to figure out dependencies, it's going to be helpful to be able to look at each line of the assembly and figure out what registers that line is reading from, and what registers that line is spreading to.

And there's also information in the assembly file I went into that last time, which tells you kind of for each chunk of assembly what the corresponding source file line is. All right, and the actual output of the static profiler, like I said, it will spit out a schedule of when each instruction runs. And in the schedule, you're going to see one digit for each cycle that the instruction is running. So however many digits there are, that's how many cycles are being used.

And in front of some of the instructions, you're going to see these dashes which represent stalls. So these three dashes in front of this instruction fm means that based on the pipeline, you could've scheduled this instruction up here right after the previous instruction finished. However, because you're waiting on particular dependencies, you're only going to be able to schedule it these three cycles later. So that's a three cycle stall.

And the static profile output is also going to show other information. The original assembly appears on the right-hand side. And which pipeline the instructions are going into is going to show up on the left. Any questions about this format?

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Dual-issue just means that these instructions are going to be executed at the same time.

**AUDIENCE:** So can people figure out, for example, what is the latency for fm instruction? How many circles does fm take? [INAUDIBLE].

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Pardon?

**AUDIENCE:** What are the numbers on the right?

**PROFESSOR:** Over here? All right, so these are the arguments to the assembly instructions. So remember the first one is going to be the destination register and the other ones are going to be all the source registers that that instruction is going to use.

All right, so again, we want to be able to reorder these instructions to minimize stalls. And what you're going to do is you're going to try and look for instructions that are going to take a long time potentially, for example, loads and floating point instructions are going to take a long time. And of these instructions have dependencies then those dependencies are going to stall.

So not only do you want to minimize stalls, you also want to dual issue instructions whenever possible, although that's a little bit more difficult to just kind of eyeball. But anyway, if you get up to the ideal running point, where you're able to dual issue every instruction, then that means you're going to be running two instructions per cycle, or it's going to take 1/2 cycle per instruction.

Anyway, going through these assembly files, and figuring out where the dependencies are, which instructions you can move around, it's kind of tedious, and you're not going to be able to apply these techniques to your entire program. So if you only have limited time, what you're going to have to do is try these on the most critical code first. And that's usually going to be loop bodies, because those are

going to get executed many, many times. And any savings that you see there will be multiplied many, many times over.

So I'm going to go through an example of one possible optimization you could do. So we'll actually go through these same files in the exercise about two minutes from now. This is from Lab 1. And this is one of the code snippets or the assembly snippets from the SPU program.

So notice that over here we have these load instructions followed by a, I guess this is a shift instruction. And there's two pairs of these. And in each one, the shift instruction is going to depend on the result of the previous load. Does everyone see that from the arguments? So now the problem is that because the shift is happening right after the load, it's going to stall for a long time, because the load takes six cycles.

All right, so what can we do over here in order to reduce stalling? Yep?

**AUDIENCE:** Two loads concurrently?

**PROFESSOR:** Yeah, we can do the loads concurrently, and in general, we just want to move these loads up as high as we can. So if I move the two loads up to the top of this-- oops-- the top of this snippet, then we have two loads followed by all the instructions that came after. Now the first shift is going to stall by only two cycles, and the second shift is not going to stall at all. All right?

**AUDIENCE:** [INAUDIBLE]?

**PROFESSOR:** So, pardon?

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** So the question is what would this change look like in code? The thing is that kind of the order of these generate instructions is going to be dependent on the compiler, and it's really up to the compiler how it wants to order these instructions. So this change is not going to be reflected in like your C code. And that's why you have to do these optimizations after the assembly is generated. Does that make sense?

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** If you have a smart compiler then it might do some kind of instruction scheduling in order to minimize stalls using this methodology. Yup?

**AUDIENCE:** I'm quite impressed with XLC. I compiled the same program under GCC for SPU with optimization level 3 under XLC. XLC ran in half the time. So it was doing a much better job of instruction scheduling--

**AUDIENCE:** Than GCC?

**AUDIENCE:** Than GCC. So that's something, if you've got a choice, and the XLC compiler is working for you--

**AUDIENCE:** In general, this kind of sort of low-level assembly hacking for instruction ordering is not something you'll do, just because compilers are getting incredibly good at doing instruction scheduling. But here the exercise is for you to sort of get familiar with the tools and understand a little bit more in detail as to what's going on at the instruction level [INAUDIBLE] pipelines.

Another thing to note here is, well, two things you could consider here, how high up can you push the limits? And related to that is, why can't you completely hide the latency for the first load, and hence the first shift still takes two seconds to stall. So that's dependent upon how much slack do you have in your schedule, dependent on dependencies and where the loops starts.

So in some cases, you just simply can't hide the entire latency. So it's part of considerations you have to consider. But [INAUDIBLE] and the compilers are getting really good at it.

**PROFESSOR:** And in case you're wondering, I think this code was compiled using XLC on no optimizations, which is why there's this really easy opportunity for optimization here. It turns out, if we run the timing utility again, we saved eight cycles. So notice that on the previous picture, we're going up to seven in that column. And now we're only

going up to the previous nine, all right?

So what we're going to do for the exercise right now is you can try something like this. What we're going to try and do is just improve performance by rescheduling instructions. And so if you download the tarball, you'll get the exact same file I was working on before. And what we're going to have you do is just generate the assembly, practice generating the assembly, modifying it, and then continuing with the rest of the build process to get a new object file that's based on your modified assembly.

All right, any questions? Now you just need to find, for example, one opportunity for optimization. And you can do the exact same thing that I did, if you want. And so the key is just after you've done your reordering, you want to re-run the timing utility to see how many cycles you saved. Also you want to, of course, continue the build process to get the final executable file. And you want to run that just to make sure that your code is still correct.

**AUDIENCE:** Are we able to get this?

[SIDE CONVERSATIONS]

**PROFESSOR:** OK, so this static profiling process does have its limitations. The first is that, and this is a pretty major limitation, the static profiler assumes that none of the branches are taken. So basically it just zips through the entire assembly file in a straight line. So that means you're going to have a skewed view of how long code inside conditionals or loop bodies is going to take, because conditionals, if there's savings in there, they may not run at all. And loop bodies, any savings in there may get multiplied many, many times.

So if, for example, you see that you've saved eight cycles in the static scheduling, you're going to need to know some more context, that is whether it's inside a loop body and how many times you expect that to run. So you're going to need a little more information to get a good idea of how these instruction reorderings are actually affecting the runtime of your program.

And also, because this is static profiling, which means you're not using an actual program run to get data, you're not going to get behavior-- you're not going to capture any behavior that depends on the inputs to your program. So for example, you don't get counts of how many times each loop runs, or when each branch is taken.

One other thing to worry about, which isn't going to be manifested in these static profiling predictions, is branch prediction. Now remember the static profiler is just going to ignore all branches. And the thing is that branch prediction, because branch prediction is used to reduce the stall after a branch is taken by kind of pre-fetching instructions and possibly doing other things. So what we'll do is guess the direction that the branch will take. And then we'll start pre-fetching instructions from the expected place where it's going to resume after the branch.

And the thing is, if your branch prediction is wrong, then you're going to incur a pretty large penalty. And the penalty is on the order of 20 cycles while it flushes the pipeline and starts fetching instructions from the new place. On the other hand, if branch prediction is correct, then there's no penalty at all, actually. So the next instruction will just resume right after the branch.

Anyway, you can give hints to the compiler to tell it what you think the outcomes of branches will be. And this can help in some circumstances. Basically there are these macros that you can define. And what you're going to use is this built-in expect compiler intrinsic, and what that means is if you put that inside an if, it will mean, you know, run the if as if the condition were exp. But then I'm going to expect that the value of this condition is going to be, for example, true or false. All right, does that make sense?

Anyway, just one note about the exercise that you did. I believe the original run time was 469 clock cycles. Actually, if you run XLC with 05 optimizations, it will reduce that to 188, so almost 280 cycles of savings.

All right, so we'll-- pardon?

**AUDIENCE:** The one that we just [INAUDIBLE].

**PROFESSOR:** No, so the original code was-- the original assembly was compiled on O 0, or no optimizations. But then if you're doing an optimization by hand, it saves eight cycles off of O 0. These are actually pretty easy to find. And the 05 optimization level will actually shave off 280 cycles.

So we'll break for about 10 minutes, and then after the break, we'll do SIMD on Cell.