MITOCW | Recitation 4: Cell debugging tools | MIT 6.189 Multicore Programming Primer, IAP 2007

PROFESSOR: So Phil is going to do the next three recitations. So the first one today is really going to focus on how do you actually debug programs on cell. How many people in here have used gdb, or are familiar with gdb? OK, good. So there's going to be a mini tutorial on how to run programs and attach gdb, which is a debugger that's commonly used for debugging programs. We'll show you how to debug PPU and SPU programs. And he's going to talk a little bit about some other things that might help you in terms of performance debugging, finding out where things are not going as well or as fast as you might expect them to.

And the next two recitations, which will be next week, will focus on doing some actual performance-specific optimizations. So you've written your program. It's working. Well, the functionality is there. But it's not performing as fast as you might like. Or you think you can improve performance a bit better. So what can you do? So he'll show you some tricks and some things you can do to get the performance up.

For some of you who have asked about the Eclipse debugger, the Eclipse IDE, it does run reasonably slowly on the PlayStation 3 because of memory constraints. So if you have lots of people trying to use it on the PS3, it'll probably be unusable. So for those of you who are interested, we'll just do separate tutorials offline. Because I don't know how many people will actually end up using it. Other than that, [INAUDIBLE].

PHIL: All right. So today we'll talk about how to run gdb on Cell. And then we'll also do some static profiling where you can figure out when instructions are going to be executed. And then I'll talk a little bit about the dynamic profiling tools that we have available. So you can use gdb to examine the state of your program. And this can help you figure out bugs that you might have. There's two ways that you can use gdb. The first way is when your program crashes, you can figure out what was happening at the time that it crashed. Another way that you can use gdb is to kind of run it from the beginning. You attach gdb to your program, and then you step through all the things, all the instructions that are being executed while the program runs.

In order for gdb to be able to provide useful information about the state of your program, you have to give it a little help. You need to compile your program using gcc -g. xlc -g will also work. This puts in some extra information into your program so that when gdb is looking at the state of your program, it's able to figure out what line numbers in the source code that corresponds to.

So you can just add dash g manually to your GCC invocations, or you can also set this compiler-- or you can set this option in the makefile, CC_OPT_LEVEL and set that to CC_OPT_LEVEL_DEBUG. And that will do the same thing as adding dash g. All right?

Now we have a special version of gdb available for use on the Cell. ppu-gdb is used for debugging ppu and spu programs after they've been linked together. And how you invoke it if you're going to run your entire program using gdb is you type ppu-gdb and then the name of your program. And what you get is a gdb prompt where you can issue additional commands to the debugger to control execution of your program, and also ask the debugger to tell you various things about your program.

One thing you can do is export SPU_INFO=1 before you start your program. And that will print some extra debugging information about when threads are being created and destroyed.

Like I mentioned, you can also use gdb to attach to a program that's already running. And if you want to do that you still need to provide the executable name. And you also provide the process ID of the program to attach to.

PROFESSOR: Do people know how to get the process ID? [INAUDIBLE] How many people don't know? The easiest way to do it is if your program's running just type "top." And that'll actually show you what's running on your machine. And there'll be a column that has a process ID. Another thing you can do is just type "ps." We'll add those to the recitation slides. [INAUDIBLE].

PHIL: So I forgot to mention. If you're invoking your program with gdb, then after you type ppu-gdb with your executable name, gdb is just going to sit there and wait for instructions. And what you can do is type "run" and that will run your program normally. But the difference is that gdb will pick up and if something goes wrong. And then that's where you can step in and look at the program state.

So what kind of things can you figure out about your program? Anytime you have this gdb prompt and your program is running, you can type "bt" to get a stack trace for your program. And this will tell you which functions are calling which functions. It's going to give you a list of frames. And the top one is going to correspond to the deepest level of function nesting, or the deepest function call in the program. And of course, the last one on the list is going to be the main or the first function that was called. That make sense?

And for each stack frame gdb will tell you the name of the function. And if source code is available it will tell you also which file the function is running in right now, and at which line number. And so using that information you can figure out exactly where it was that this function called the next function and where was that each function called the next function after that.

Now whenever you've halted the program state, you can get information about what the local variables are. And to do that you can type "info locals" and it will print a list of all the local variables and their values. gdb, because it gets annotations from the compiled program, it knows about what data types are associated with which variables. And so it will be able to format the output appropriately.

So for example, it knows that i is an integer. Whereas if you had i as, for example, a floating point number, it would print that floating point number representation instead. Instead of looking at single variables you can also ask gdb to evaluate arbitrary expressions for you. And this is actually a really powerful facility. You can pretty much type any expression that you would be able to use in C or C++.

When you type "print VARNAME" gdb will look up that variable name for you. But sometimes if you have more than one variable available in your function under the same name, for example, these variables are in different files or different functions that are all available, then you may need to disambiguate the variable name. And to do that you just proceed the variable name with either the file name that you want to use or the function that the variable occurs in. Any questions?

OK. And because gdb knows which line numbers the programs is executing right now, then it's able to show you the source code corresponding to the current place in the program. And in order to browse the source code, if it's available you just type "list." In general whenever gdb has stopped in a certain place you can type "list" to get a listing of the source code near that location. And what list will do, by default, is show 10 lines of source code near where the program stopped.

If you type "list" again it will show 10 lines following that. And you can keep typing "list" to browse the source code. You can also get the code that's at a particular line number if you want to look at another place in your program.

All right, so this is how to use gdb to actually control the flow of your program. If you invoke gdb to start up your program, then you can type "run" and that will run your program with a normal flow of execution. And it will only stop when errors occur. But you can use a couple of these commands to actually go step by step through your program. And wait for one line to be executed, and then decide whether you want to look at the next line or not.

If you want to do this kind of step-by-step execution you start by using the start command instead of the run command, which will run continuously. Then anytime you want to jump to the next instruction, you can use next to get to the next line in the current procedure. You can also use step, which, if there are function calls on the current line, will descend into those function calls and show you what's going on over there.

And if you're inside a function call you can type "finish" in order to jump back to right after the function call ends in the caller of that function. And if you want to stop line-by-line execution you can type "continue" and it will just continue without interruptions. Any questions? Basically any time you use one of these line-by-line commands gdb will show you the line that you're on. And if your program just jumped into a new function or something, then it will also show you what function and what file you're in. So this is very helpful for following what's going on in your program.

OK. So in addition to just running through your program step by step, you can also choose to only stop your program when certain things happen. So this is very useful for debugging certain pieces of code where the code is very deeply nested inside and you don't want to step all the way to there. You just want to stop when you get there.

And gdb allows you to define breakpoints, which are places in your source code. And once gdb gets to a point of execution which corresponds to that point in your source code, it will stop and ask you what to do. You can set a breakpoint that's associated with any particular function in your source code or any particular-- well, what you can do is if you do break function then it will create a new breakpoint. And it will stop execution any time you get to the beginning of that function. But you can also set break points in the interior of a function by using line numbers. All right? Any questions?

Now sometimes you want a little bit more flexibility than that in setting breakpoints. If you only want to set a breakpoint to happen when a certain condition is true-- this is helpful if you're trying to track down a bug-- then you can write "break" and any of these things, and then "if expression." And again, you can use any sort of expression that you'd be able to use in C.

And what gdb will do is every time execution runs up to that line it will evaluate the expression. And only if that expression is true will it stop to ask you for control. At any time you can see which breakpoints are active by doing info breakpoints at the gdb prompt.

And you can also remove breakpoints at any time. When each breakpoint is created it's given this number. And the numbers start from one. And to remove a breakpoint you just do remove followed by the number that was assigned.

In addition to breaking, you can also set a watchpoint. What this does is it will halt your program every time a particular value changes. And this is useful for tracking down certain kinds of bugs. If you're trying to figure out where a certain value for a variable came from, there could potentially be many, many places in your program where that value is set. And you don't want to have to set a breakpoint for all those and watch them. So you can instead set a watchpoint which will just tell you when that value is set.

Watchpoints work a lot like breakpoints. If you do info breakpoints it will also list the watchpoints. And I believe you can also remove them the same way with remove.

All right, so I mentioned how you can examine your program state by looking at the values of local variables. If you want a more low level view of what's going on in your program, you can actually look at the raw memory. And gdb allows you to specify an address. And it will tell you exactly what data is stored in the memory at that address.

But because there's not necessarily annotation info telling gdb what kind of data is stored at that address, and of course, there's multiple ways you can interpret any particular piece of data, you'll have to tell gdb exactly how you want that data to be interpreted. So you can interpret any particular block of memory as, for example, a series of machine instructions. Then gdb will tell you what the instructions are as if you were looking at an assembly listing.

You can ask gdb to interpret the memory as if it were an array of integers and print them out either in hex or in decimal. You can ask gdb to display the data as if they were addresses or floating point numbers. And how you do this is you type x for examine memory and slash. And then the number following is the number of words you want to look at. And then the letter corresponds to one of these letters and tells gdb how to format the output. And then the address will be the starting block of where you want to start examining memory. All right? Any questions about this?

OK, so I mentioned that at any time when you stop your program there are going to be multiple function calls which are active, corresponding to main, which called this other function, which called this other function, which called wherever your current point of execution is. And you can actually examine the state for all of these stack frames separately.

When you do bt it's going to give you the list of frames. And they're going to be numbered from 0 to however many there are. And you can jump to any of them by using frame and the appropriate number. And it's going to default to frame zero, which is the closest to where your program is actually executing. But you can examine the state in frames which are further away.

So that means when you're evaluating variables, each variable only makes sense in the context of a particular frame. And you're able to go up the call stack to figure out what the value of a particular variable is in this function, which called this other function. You can also use the commands up and down to just jump to the immediately adjacent frames. All right?

OK, so you can use gdb from emacs as well. And emacs provides a really handy interface for gdb. If you do M-x gdb emacs will invoke gdb for you. And when you do this, you're going to want to tell emacs to use ppu gdb instead of regular gdb. And emacs will just ask you what you want to invoke. You just want to replace gdb with ppu gdb.

Anytime you're debugging within emacs, if emacs has your source code files open, it will show you the current point of execution by drawing a little arrow next to the particular line in the buffer. You also get a bunch of keyboard shortcuts that you can use to do some common operations. You can set breakpoints with Control X Space, just by placing your cursor at the particular line that you want to stop at.

So you don't have to go to gdb and type break whatever whatever. But when you do m-x gdb in emacs, you get a separate buffer for gdb. And so you can issue all the commands that you normally would want to.

OK, so the first thing we're going to try-- this should be really quick-- is we're just going to open up a brief program and make sure you can invoke gdb on it. And you'll have to set Cell top if you don't have that set already. I'm just going to do it on the same one. [INAUDIBLE].

AUDIENCE: How do you exit gdb again?

PHIL: Oh, you can do Quit or Control D. So once you've attached gdb to a program, when you exit gdb it will quit the program that you were running, unless you do detach first. For this one it doesn't really matter, because we're looking at a crash anyway.

PROFESSOR: [INAUDIBLE] Anybody [INAUDIBLE]? OK.

AUDIENCE: How do you evaluate something [INAUDIBLE]?

PHIL: You can evaluate an expression using Print. OK, any questions? All right, so all I did here was run and then the program crashes. This is just lab one without the alignment in the control block that you need to make everything work correctly. And so all I did here was run and then the program crashes and then print cb. And of course, this thing is going to be 0 because the DMA transfer to do that doesn't work. Questions?

OK so we can exit the debugger with Control D. And it'll ask if you want to kill the program.

So gdb also has features to help you with debugging programs that have multiple threads. Whenever a new thread is created or a thread is destroyed, gdb will print a brief message to tell you. And one important thing is it will print this LWP number on the PlayStations.

To get a list of threads you can type info threads. And gdb always maintains this thing called the current thread. And by default, when you do many of these other actions they're going to apply to the current thread. And so to examine things about other threads you're going to have to change the thread. And to do that you do thread and some number.

When you do info threads, it's going to give you this list of threads which are numbered, for example, 1, 2, 3 on the left. And the LWP numbers that are displayed are going to correspond to these that came up when the threads were started. But you're going to have to use these numbers on the left to switch between threads.

And when you do info threads, gdb will mark the current thread with a star on the left. And it will also show you where each thread is executing right now. Questions?

AUDIENCE: Where does it show you [INAUDIBLE]?

PHIL: It will show the procedure and the file name and line number, if that information is available.

PROFESSOR: If you can't find out which [INAUDIBLE].

AUDIENCE: That's what I was saying--

PHIL: Ah. No.

AUDIENCE: [INAUDIBLE]

PROFESSOR: I don't believe you can get that information. We can check. David, do you know? Sorry [INAUDIBLE]. So David says maybe you can do it on a simulator, but not [INAUDIBLE].

PHIL: OK. So we're going to try this brief exercise, which is just to get you to work with dealing with multiple threads. And what you're going to do is load the lab 1 program, which is the correctly working solution for lab 1. And if you'll recall, this program has multiple threads. The PPU thread maintains an array of control blocks. And it's going to send the addresses of those control blocks to the SPUs. So what we're going to do is we're just going to set breakpoints at the right places to verify that the first SPU thread is getting the control block which is the same as one of the control blocks in the PPU program. All right, any questions about that? This is in [? rec ?] 4, lab 1.

So you're going to have to run and set breakpoints at the right places. And you're going to want to set one break point in each thread to be able to examine the value of CB that's being produced.

PROFESSOR: [INAUDIBLE] right after the recitation. As another exercise for those of you who did get through, the way you'll actually run into these problems is you'll run your program and you might end up with a bus error. So what you might want to do is then launch gdb. And you run to the error. And then you trace back in the execution to see, uh-oh, is my control block value matching? So you might want to try that for the second exercise online.

PHIL: Should do the stack profiling thing?

PROFESSOR: Yeah, just go through that. [INAUDIBLE].

PHIL: OK. So one problem that you may have run into is that gdb will remove breakpoints from threads that exit, which is a problem if you have more than one thread running the same program. And if you're debugging SPU programs, gdb may complain about not being able to find the source files. But if you just ignore that message, it seems to find them OK, I think. You can use SPU gdb to debug the SPU programs by themselves. You can try that sometime.

OK, so actually figuring out what the errors are, unfortunately you don't get very much information from the actual errors that occur. But maybe looking at where the errors are occurring can help you figure things out. If you've run into memory misalignment problems or the problem in the last recitation where we had DMA transfers that were too big, those are all going to be bus errors. Under various other circumstances you might get segmentation faults. And if you think your program is running into a deadlock, you can also use gdb to kind of attach to the program and then examine the state of that program to see what's waiting for what.

OK, so static profiling. We have some tools for the Cell that will allow you to, when you have a sequence of assembly instructions, figure out when those instructions are going to get scheduled and how fast the resulting sequence will run. So in order to take advantage of this you need to use GCC dash big S to generate your assembly. And you can also do that with xlc.

If you're using our makefiles you can also use make whatever file named dot s in order to generate the assembly from your source code. Then once you have the dot s file you can run this utility called SPU timing. And what SPU timing does is it will take all the instructions that are in your assembly and figure out the dependencies between them. And then it will figure out when the earliest point is that each instruction can get executed. And it will print out the schedule that's generated.

So if you provide the dash running count option, then it will also show you how many cycles in all the entire program will take. And the output goes into file name dot s dot timing.

So for how instructions are scheduled on Cell, if you'll recall, there's two pipelines for different kinds of instructions. And some instructions can only run on one or the other of the pipelines. And some instructions can run on both. Now, how Cell actually schedules those instructions is it's always going to go in order that the instructions are specified in the binary.

And whenever there's two instructions next to each other and the first instruction can run on pipeline zero and the second instruction can run on pipeline one, then Cell will try and schedule those at the same time. And that's called dual issue. So if you were taking advantage of that dual issue all the time, then potentially you could schedule two instructions every cycle.

Oh yes, and unlike a lot of other architectures nowadays, Cell does not have dynamic rent branch prediction. All the branch prediction is encoded inside the assembly that you're using. So that means for any type of loop or whatever, you have to be sure to get the branch prediction right. If the branch prediction is wrong, Cell is going to end up pre-fetching instructions along the wrong line. And it's going to have to stall by about 20 cycles when it figures out that the branch is wrong.

So if you're looking at the generated assembly on Cell, all the instructions are going to be of the form operation, then the destination register, and then two or more sources. And if you're trying to just kind of orient yourself in the generated assembly, there are sometimes these helpful markers.

So if you're looking at the generated assembly for dist_spu, then it's going to have a header at the top which says which files were included inside this file. And then where the actual assembly is there'll be these markers that say, for example, location 1 19. And what this means is that here is the assembly corresponding to file 1 line 19. So that's kind of helpful if you're trying to get your bearings inside the assembly. Because otherwise it's really hard to make heads or tails of. Questions?

All right, so after you actually run your assembly through the static profiler, it will spit out this schedule. And what the schedule shows is which clock cycles are used for every single instruction. There's one line for each assembly instruction. And what it's going to do is it's going to print one digit in this schedule for each cycle that the instruction takes. All right?

So you can kind of think of this dimension as the passage of time. And what happens is that these guys, when they get to the right-hand side, they'll wrap around it 50 columns or whatever. And you'll be able to notice when these instructions are being scheduled. And sometimes when there's dependencies between instructions, the instructions are not able to get scheduled at the earliest possible time. And when that happens you'll see these dashes, which mean that the instruction is being stalled to wait for one of the dependencies. In order to make your code fast you're going to want to eliminate these stalls. And you can do that to a large extent by reordering your instructions.

AUDIENCE: Can you stop at the previous slide?

PHIL: Yep.

PROFESSOR: [INAUDIBLE]

PHIL: Pardon?

PROFESSOR: The point of instruction scheduling [INAUDIBLE].

PHIL: So the point of instruction scheduling is going to be to minimize the number of stalls. And you can do that by, if you have instructions which are going to be dependencies for other instructions, you just want to move those as far up as you can.

PROFESSOR: So ideally you would get to instructions per cycle, or how many instructions can you use per cycle? Two, because you can have dual issue, two pipelines. Or how many cycles per instruction [INAUDIBLE] by a half. Because you're getting two instructions per cycle, so [INAUDIBLE]. So we have an exercise that actually has you understand the assembly code, and then doing the instruction reordering.

So we'll leave that on the slides. You can do it offline. But we'll pick up with this next week and go over instruction scheduling, some DMA tricks for improving performance. And in particular, the thing we'll focus on quite rigorously is [INAUDIBLE]. So Phil's going to walk you through how you actually [? synchronize ?] and get performance from the vectorization and the intrinsics. We'll talk a little bit about [? heat ?] dynamic profiling.

And for those of you who are still having problems with gdp, we'll try to resolve those now since it'll probably be very useful for your projects. And we installed CVS in the main directory on every PS3, for those of you who actually want to use it and go through the trouble of setting up their own CVS. The CVS that's satisfied actually notified the local users on that machine. So that would be everybody on your team. It will send out an email whenever somebody does a check-in or an import, to let them know that there's some new information that they don't want to update. If you don't know how to use CVS or want a quick tutorial, just stop by. Some of the TAs will be able to help you. OK? [INAUDIBLE]