ARVIND THIAGARAJAN: So I'm Arvind and this is Micah. And we worked together on a framework for flexible stream processing on the cell. So we were also oriented towards an application which is, in this case, a software radio. And we used this application as a case study. Where is the mic?

I'll try and speak as loud as I can. So our project was about flexible stream processing framework for the cell. And we used the software radios as a case study, essentially, of an application you could build using this framework.

The motivation for our project is essentially what we've been discussing in class. It's been reiterated. The cell isn't easy to program. There's no shared memory. There's just message passing, which is quite messy. You have to explicitly parallelize your programs if you want to write them as, say, custom C programs.

Some of the groups have describe the challenges they faced when doing that. Extracting parallelism can be tricky. For example, if you want to do pipelining, then on DSPs you have to predict what addresses you're going to require, and set up a DMA so that those addresses are fetched in advance. And as Bill mentioned in his talk in class, stream programming can help alleviate some of these issues for some applications, like software radio where the applications fit naturally into the streaming model.

So what we tried to do for our project was build a light weight-- as light weight as possible because the code has to fit on DSPs, but as expressive as possible, as well, so as to simplify life for developers. The streaming framework which is targeted specifically at signal processing applications. The data model, at least, is based on a research project that I have worked on in the past, which is a WaveScope streaming

database management system. It is is a research project with the database group.

The data model is essentially an extension of the streaming model to handle larger blocks of data so as to process them better. For example, several signal processing operators, like the fast Fourier transform, need to do multiple passes over the data and, therefore, trying to treat streams on a sample by sample basis leads to high cost, scheduling overhead, as well as inefficiency.

So that's what the data model does. And we tried to port this data model over to the cell processor and see how well we could exploit the features of the cell processor. In particular, the high on-chip bandwidth between the SPEs to do with streaming.

So our case study was, as we mentioned, a simple software radio application. It's really simple. It uses incoherent demodulation, as well as just simple amplitude-shift key modulation. As I said, the main goals of our framework were to simplify life and modeling as much parallelism as possible-- try and extract pipeline parallelism, data parallelism. Some of the kinds of parallelism, We'll mention. The kind of parallelism we were able to finally implement-- so far, it's only pipeline parallelism, but as future work, we'd be interested in the other kinds of parallelism as well. More about it as I go on.

So in the framework we at least implemented, the programming model is quite simple. The basic execution unit is what we call an operator. It's analogous to what in StreamIt would be called a work function, or what in GNURadio, which is a framework for building soft radios, would be called a block. So these operators can be any arbitrary C++ classes with state, and they implement an iterate method which the developer has to overload in order to process a block of data.

The WaveScope data model also provides a library for managing memory and passing blocks of signal data between these operators. Applications in this model are built by chaining operators together. So this is a snippet of some of the code we wrote for the software radio, roughly. So you create a box, let's say a FIRFilter, which processes elements of type float. And then, it takes in some arguments, initialize the filters, parameters, and so on. You want to create a white noise

generator and hook up the filter to the white noise generator. We use this to simulate a simple channel, [INAUDIBLE] channel.

So I'll just describe the components of our framework. We have a lightweight scheduler on both the PPE, as well as the SPEs. Right now, it uses a static operator mapping in the sense that you have to specify a static configuration file, where you say this operator name will run on this [? SPU ?] number.

But we've not completely implemented dynamically reconfiguring at runtime. And we haven't yet seen the need for doing that. So it wouldn't be too hard to add if needed. But right now, you can easily shuffle around the operator mapping by tweaking the configuration file.

Signal blocks, as I said, were adapted from WaveScope. They use reference counting and avoid in-memory copies, which can be quite expensive, especially on the cell. They also provide a convenient API to manipulate signals. So you don't have to do much of the memory management yourself or debug any of the hard problems to do with memory management. We also ported this library to ensure that data is aligned for you automatically and transported via queues.

So one of the major things we had to implement was a queuing library and remote heap management-- what amounted essentially to a remote heap management library. So in some sense, we faced a choice here. We could either have the PPE control and allocate memory statically and make all the decisions about what memory is allocated where. Or, we could have the SPEs manage it themselves.

We decided to go for the latter, partly because it was more dynamic and, also, because we weren't sure what the implications of all the control flow passing through the PPE were for this. So we chose the second approach, which is autonomous memory management.

So when an SPE sends a streaming data element to another SPE, it doesn't have to actually explicitly request the other SPE for allocating memory. It has a remote heap interface so that it can directly allocate data and write to the SPE. This is currently of

fixed size, but it could be improved by using this heap to share a bunch of queues.

Right now, we have one remote heap for each queue between operators on different SPEs. So our system automatically handles pipelining streaming data from SPE to SPE using the DMA API. So Micah will takeover from here and describe the software radio implementation briefly.

MICAH BRODSKY: So our software radio implementation is relatively simple. We weren't pushing on that too hard, especially because it took the vast majority of our time just to get the framework to work. Saving the programmer trouble meant that we inherited a lot of trouble ourselves. It breaks down to about 25 boxes, which we took in our config file which is manually mapped to the SPEs. About 3,000 lines of code, most of which is framework.

So I guess, if we were more put-together, we'd have a nice diagram to show you. There's enough time. I'll try to draw a quick diagram on the chalkboard. We're simulating both sender, receiver and a channel.

So the computation in question looks something like you have a bitstream. You need to take these and convert them into some-- can you read anything? So you take stream of bits in. Take bits. Basically, use a lookup mechanism to convert it to an analog waveform, and filter that to produce something that has a narrower spectrum.

Running out of space here. Means that. Multiply that against a sine wave.

ARVIND THIAGARAJAN: That's for modulation

MICAH BRODSKY: And so you get-- you've probably seen pictures of this. It's very much like AM It's basically binary AM. [INAUDIBLE]. It's one of the simplest things you can do.

Then, this is to simulate a channel, which is a random FIR filter, finite impulse response. What that means, it's basically taking the copies of the input at different time offsets with random coefficients and just summing them up. It's a huge multiply

add computation.

AUDIENCE: [INAUDIBLE]

So this is 80 taps. And add some Gaussian noise. And then this, we take over, and then try to figure out what we put in in the first place. So a bunch of filtering. Again, more finite impulse response filtering.

There's a little closed loop that tries to estimate the signal amplitude and correct for it. That's called automatic gain control to sort of keep it constant.

And I'm probably getting these things a little out of order. This is the incoherent demodulation part. We square the signal to get rid of the carrier. Automatic gain control. Filtering.

There's another loop. This is called a phase lock loop. The idea is try to match a sine wave to some input signal. I don't know how to explain it very well.

It's basically a locking type of detector. The idea is to lock into the phase of some periodic thing. This is for recovering when do you sample. Because you've got this messy waveform, you've got to know when to look and, say, OK, is it high, is it low to get a bit out.

ARVIND THIAGARAJAN: [INAUDIBLE].

MICAH BRODSKY: Yeah. I think that gives the picture. I'm probably boring everybody. Here's a picture generated from the system. So the green line is the data in. Hi, low. [INAUDIBLE]. The red line is the analog signal out right before it's supposed to decide what the heck the input was. This is after squaring, and filtering, and automatic gain control, and all that.

The little blips are actually because we used a modulation called alternate-mark-inversion. It basically flips every one. That's why it's blipping instead of being constant, which is to be [INAUDIBLE]. And then the little blue daggers are the

results of the phase lock loop to try and recover when to sample. And they're kind of off, but they're kind of right.

And so, if you take the little blue blip, if the red line is above 0, it's a 1. And if it's below 0, it's 0. And that's how you get your bits out.

This was hard to get right, mostly because of the framework issues. Implementing distributed objects on a system without real shared memory is hard because you have to serialize everything into a stream of bits and deserialize. So it really makes pie out of any existing object oriented code.

We did quite a bit of work to get decent lock-free almost zero-copy. Another day or so, we probably would have gotten zero-copy-- transfer-- streaming of the data from place to place. And we had to keep the code footprint low.

C++ is bloated. We don't have an overlay system yet to-- if SPE is not running a particular box, it still has to have the code for it. So we don't have any infrastructure for--

ARVIND THIAGARAJAN: There's some macros to get around that, right?

MICAH BRODSKY: Yeah. It's pretty messy. So all the code is on all the SPEs. So code bloat is particularly a issue. And XLC has this particular penchant for runtime type information and exception handling. Incredible amount of voodoo is necessary to get them and that 70 K of useless bloat out of there.

ARVIND THIAGARAJAN: We pretty much have the--

MICAH BRODSKY: It works, but not always.

ARVIND THIAGARAJAN: We did manage to get it running long enough to get some measurements.

MICAH BRODSKY: Yeah, we did get some decent data out. Running on the PPE only, we can about

170,000 samples per second through. And with the scheduling file, that's kind of rule of thumb-- we just roughly said, OK, that looks about this big. We'll throw it on this SPE, SPE, SPE. We got roughly four times that using five SPEs.

ARVIND THIAGARAJAN: The core foorprints are really large, but--

[INTERPOSING VOICES]

MICAH BRODSKY: We really had to push down things like our queue. We just didn't have enough memory. It sucked. We basically just said that already.

ARVIND THIAGARAJAN: Some performance bottlenecks, I guess.

[INTERPOSING VOICES]

MICAH BRODSKY: Interesting performance behavior. We found that the SPEs are ridiculously underutilized. Most of the algorithms are quite a bit zippier on the SPEs. And so, they may be running about a third of the time, and the rest of time just waiting for input.

And then the PPE, which is doing only a tiny amount of the computation-- basically just feeding it in and sucking it out-- is spending half of its time all busy and, the other half of the time, stuck in flow control waiting for queue space, which is-- our flow control algorithm sucks. Better with time. So it seems like you should be able to do quite a bit better than we did with a bit more work.

We need to cut down the footprint. And once we have a little bit of breathing room and get rid of the nasty race bugs and such, we can finally investigate what was our original, pie in the sky goal of automatically deciding what goes where, taking measurements of the performance and then feeding that back to producing a better placement of operators, and applying data parallelism by instantiating operators on multiple different SPEs and splitting the data stream. And just doing more.

AUDIENCE: Since your PPE is at 40% to 50% utilization, did you put actual work on there?

MICAH BRODSKY: We did put some work on there. Actually, we put work on there because we couldn't fit all the boxes-- the code for all the boxes under the SPEs. So we started strategically to put a few things at the beginning and a few things at the end which weren't supposed to be very computationally intensive, and yet they managed to take up half the CPU.

The issue with the other half of the CPU is that it's actually inside an inner loop deep recursive in blocking because, basically, our back pressure isn't online yet. So if it tries to emit something to a queue, and that queue is full, it just stops. And it really could be running a whole bunch of other stuff, but it's not smart enough to do that ahead.

AUDIENCE: [INAUDIBLE]

MICAH BRODSKY: Unlike a model like StreamIt, everything here is asynchronous and code driven. The SPUs they can decide on the fly how much to emit. The programmer doesn't have to declare anything, But it means everything's asynchronous. And so you basically [? race the issues ?] galore.

AUDIENCE: But in this a application, do you find any dynamic rates?

MICAH BRODSKY: In this application, there's not much. It's a very simple application. If we actually went to packetization, error correction, compression, things like that, we'd probably see a lot more of that. This is definitely underutilized in the asynchronous capabilities of the system.

AUDIENCE: In the case of radio, wouldn't it be OK to draw [INAUDIBLE] frame into audio data just because you're spending so much time waiting, it'd be better just to relieve pressure on the queues by just dropping some frames that are unnecessary.

MICAH BRODSKY: It might well be.

AUDIENCE: And interpolating it in the end to try to fix it up a little bit.

MICAH BRODSKY: It might well be. We decided not to do that as a part of the framework because [INAUDIBLE] policy decision, and we didn't want to make that for all possible applications. But if we figure out a good way to dispose that, that definitely would be an option. Just drop a few packets. Drop a few samples.

AUDIENCE: So what about buffer sizes? Do you declare the buffer size as well?

MICAH BRODSKY: Yeah. The way we have it now-- there's actually just a #define in the code that says all buffers are this size.

AUDIENCE: Do you need any double buffering in there?

MICAH BRODSKY: You don't need it. Well, actually, the way it works is that there's this remote heap input queue. And an upstream SPE just DMAs things in as it feels like. And then the downstream SPE looks. Is there something here? Grab it, use it. So it just works, however much buffering there is.

ARVIND THIAGARAJAN: The ring buffer and the [INAUDIBLE] should get that block [? free. ?]

MICAH BRODSKY: That's a benefit of the asynchronous approach. It just works, if you have memory, which we don't. Queues are tiny.

SPEAKER: [INAUDIBLE] is angrily telling me that his connection dropped but he's not picking up the phone. Yeah, I could see it in there.

AUDIENCE: Any other questions for them? Did you want to do the demo?

MICAH BRODSKY: Nah.

SPEAKER ON PHONE: What did you do to get race conditions?

MICAH BRODSKY: How did we manage to get ourselves race conditions? Well, it's a mix of the fact that all the SPUs can essentially operate as independent threads and are sort of asynchronously DMAing things to each other and somewhat poor-- legacy driven

architectural decisions on the way the PPU code works. Because we ported a lot of code from the WaveScope platform.

[? ARVIND THIAGARAJAN: Which wasn't very ?] well documented.

MICAH BRODSKY: And that was intended to be multithreaded with Pthreads, which in our original, naive before we actually started the class impression, we thought we were just going to port the whole thing and use Pthreads on the sub, which of course is not possible. So there were still some threaded components in there. And we introduced a lot of bugs trying to port the thing. See if i can--

ARVIND THIAGARAJAN: The surprising thing was the fact that we had to replace array constructors in order to eliminate [? RTTI. ?] And it was just a big deal

[INTERPOSING VOICES].

MICAH BRODSKY: We had to get rid of new, we had to get rid of delete. We had to get rid of array constructors. We had to get rid of--

ARVIND THIAGARAJAN: Virtual destructors.

MICAH BRODSKY: --pure virtual functions and virtual destructors. It was a mess. I guess one more pithy thing I could say about race conditions is it's incredible how many subtle race conditions and bugs we found in the remote keeping queueing library.

Because there's one-- it's a lock free asynchronous data structure. There are two threads-- two SPUs reading and writing from it at the same time. And there's all sorts of little subtleties where if you get it a little bit wrong, you end up with one of the queue pointers overrunning the other guy, and you have basically dangling pointers and things like that. At least three or four times, we spent a few hours until we discovered that was the cause of mysterious behavior.

AUDIENCE: Anything else? Thank you.

[APPLAUSE]