McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Panel Sessions of The 1991 Workshop on Multithreaded Computers

Kevin B. Theobald

ACAPS Technical Memo 30
April 1, 1993

A version of this report appears in the March, 1994, issue of Computer Architecture News

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

This document contains transcripts of the two panel sessions at the 1991 Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91.

Contents

1 Introduction

The Workshop on Multithreaded Computers took place on November 22, 1991, in Albuquerque, New Mexico, USA. It was held in conjunction with Supercomputing '91, sponsored by ACM SIGARCH and the IEEE Computer Society. Around 150 people interested in the emerging field of multithreaded computing came together to share their research and to learn from each other. Papers from the workshop will appear in a book entitled *Multithreaded Computer Architecture: A Summary of the State of the Art* (Kluwer Academic Publishers, Norwell, Massachusetts, 1994). This report contains transcripts of the two panel sessions in the workshop.

Supercomputing '91 was an appropriate site for this workshop, as multithreading is increasingly being seen as a way to improve the performance of supercomputers, especially parallel machines. In the field of general-purpose parallel computation, some areas are increasingly being recognized as fundamental issues, such as memory latency, bandwidth, and synchronization overhead. Recently, there has been active research and development in multithreaded computers, which have the potential for adequately addressing these issues. Current research directions include both architectures which have evolved from dataflow models, and von Neumann architectures with multiple instruction streams. Indeed, some researchers view multithreading as a means of combining the benefits of traditional pipelined von Neumann architecture with the latency tolerance and efficient synchronization of pure dataflow. In this viewpoint, multithreading represents a convergence of the evolutionary paths of dataflow and von Neumann computing.

These activities have attracted growing interest from university researchers, scientists from government laboratories, and computer manufacturers. By the end of this decade, it will likely be possible to implement a powerful multithreaded architecture with supercomputer performance on a single chip. As the activities evolve, both novel ideas and new challenges will continue to emerge.

The workshop was organized by Guang R. Gao (McGill University), Robert H. Halstead, Jr. (DEC Cambridge Research Lab), Robert A. Iannucci (independent consultant), and Burton Smith (Tera Computer). They invited well-known multithreading researchers to submit papers, which were then refereed. Seven of these submissions were selected for presentation in the technical sessions. There were also two one-hour panel sessions. The first panel, chaired by Dr. Iannucci, explored the architectural side of multithreading. The second panel, chaired by Dr. Halstead, focused on software issues. The speakers and panelists included representatives from both the dataflow and von Neumann traditions.

The next two sections contain transcripts of the panel sessions. These have been minimally edited, mostly to correct grammar and remove irrelevancies. There has been little editing of content. Thus, much of the flavor of the panel sessions has been preserved. Although the jokes and arguments may give this article an informal and perhaps unpolished appearance, they will give the reader a sense of "being there" at the workshop as well as a better appreciation of the current issues, debates, and research directions in multithreaded computing.

The panelists occasionally refer to statements made by others, without specifying a source. Generally, these refer to technical papers presented at the workshop. A copy of the workshop schedule is included at the end of this report. The panelists also occasionally refer to slides, which were not available for this report. Explanatory notes have been added where appropriate to try to clarify the panelists' words.

The transcripts are followed by a short (and by no means complete) list of references for systems, languages and other work related to multithreading which was mentioned during the panels. A more comprehensive bibliography of multithreaded computing will appear in the *Multithreaded* book.

2 Panel Session I: Architectural and Implementation Issues for Multithreading

Panel chair:

Robert A. Iannucci .... Independent consultant

Panelists:

Anant Agarwal .... Massachusetts Institute of Technology
William J. Dally .... Massachusetts Institute of Technology
Anoop Gupta .... Stanford University
Gregory M. Papadopoulos .... Massachusetts Institute of Technology
Burton J. Smith .... Tera Computer

IANNUCCI: The multithreading community is very dynamic (pun partially intended). We get together periodically, and agree to disagree, and learn from each other. This is another example of that. We interact so rigorously and regularly that people have been led to ask the question: what is the difference between the multithreading community and a Boy Scout troop? And the answer, of course, is that a Boy Scout troop has adult supervision!

So with that in mind, we have assembled a tentative Boy Scout troop here, and for the next hour, and only that hour, I will pretend to be an adult. And our panelists today are Anant Agarwal from MIT; Bill Dally, also from MIT; Anoop Gupta from Stanford; Greg Papadopoulos from MIT; and Burton Smith from Tera.

Now, I've offered the panelists the opportunity to make a brief position statement, and the rules are: no more than five minutes, no more than one slide, and the optical density of the slide has been prespecified. The panelists also have the option to waive their right to a position statement. So we'll go down the line, and you may waive or take your stand.

DALLY: I never refuse an opportunity to voice an opinion.

Let's start with the beginning and talk about what our goals are. Our goals are *not* to maximize processor utilization. That's a very common misunderstanding. The goal is to solve a problem in the minimum amount of time with the least amount of money.

And time has two factors. One is latency involved. Every program has parts where you don't have enough parallelism, and how fast you can run one thread is very important. Then you have parts where you have embarrassing amounts of parallelism, and the important thing is the ratio between how many ops you have to do and how many things you have to "op" them, which I'll call "bandwidth."

Cost is largely memory. After all, what we are really doing is building vehicles for delivering Japanese memory chips. And if you look at the amount of silicon that's in the processor, it's negligible in comparison.

The other goal we have is to maximize programmability. This is something which is a little bit harder to measure than time or cost. But we've spent a lot of time (in my research group at least) talking about things called "mechanisms" [8], and trying to craft them so that they can support a lot of different programming models, so however *you* think a machine is easy to program, you can do it. The other half of that, which Arvind (I think) alluded to, in how architecture affects programming models, is you have to do it *efficiently* or people won't *use* the mechanism, no matter how elegantly it reflects the model.

How does multithreading fit into this? Well, it reduces cost, not by increasing processor utilization, but actually by doing two things. It improves memory utilization, and it gives you better utilization of bandwidth.

Here's the story: Take our conventional situation, where instead of multithreading, we just decide to plop down a bunch of processors, because, after all, they're pretty cheap, and hide latency by having enough processors so that when one goes idle, there's another one to keep the network bandwidth busy. If you do this, you've replicated a bunch of 50-million-λ^2 processors, which is no big deal.

But you've also replicated a bunch of register files. And if you share the state in the register files in an intelligent way (and Peter Nuth will have more on this this afternoon), you wind up being able, in the same amount of storage, to hold a lot more register working sets, than if you give each one its own register file. So you get better storage utilization.

So really, although people talk about cache interference being a *bad* thing, you wind up being able to get by with a lot less memory at each level of the hierarchy here, than if you sum all of these together, because there's synergetic sharing of things like code, and because everybody's working set modulates over time. So you don't have to size for the worst case; if some guy isn't using very much, you gobble up some of his. You really do get better memory utilization, and *bandwidth*, which is what's expensive in these systems. You wind up not having to duplicate all these bandwidth paths and having an interesting interconnect problem between this level of memory hierarchy and your network.

So now that we've established that this *is,* in fact, a good thing to do, what are the important points to keep in mind when you do it? Well, the first is that single-thread performance comes first. If somebody wants to run a serial program on your machine, you shouldn't blow it. There's no excuse for throwing away your single-thread performance to get good multithread performance, because you don't have to do it.

So you don't want to be doing the approach where you have to wait until your token cycles completely around the pipeline before you issue the next one. Then you wind up with an issue rate which is pretty bad. Whereas as long as you don't have any dependencies, you ought to be able to pop instructions from a single thread down the pipeline and get single-instruction-per-cycle performance, or better, depending on how you organize your pipeline performance.

The next thing is register utilization, making good use of memory capacity, since that's what *cost* is, recall. (Remember that I'm talking about *cost* and not *price*. A DRAM chip and an i860XP are about the same amount of silicon area, and hence, the same *cost*; only the *price* differs.) To get good register utilization, you don't want to do a fixed mapping where you give this thread this number of registers and *that* thread *that* number, because while conventional machines tend to have things like 32 registers, the typical working set of a process is quite a bit smaller than that. And if you actually let people use only the registers they need, you can get a lot more threads in here and use the storage a lot more effectively. (The same thing, incidentally, happens at the next level of the memory hierarchy as well.)

The next point is, you want to make sure that when you switch, it is *as* efficient as possible, so that you can switch *every cycle* if you want, and not lose it, so you can handle things like remote memory references, by having your main processor switch, handle them in a cycle, and switch back. And you can do that not by having a read stage of the pipeline do your synchronization as conventionally done, examining some state bits in the register file, presence bits, scoreboard or whatever you want to call it, and then if you miss, putting a bubble in the execute stage, and having to do another read before you can get to the execute stage. Instead, if you add a synchronization stage to your pipeline, so that *only* things which are *guaranteed* all dependencies are satisfied get to the register file stage, then you don't waste bandwidth on idle cycles.

IANNUCCI: [To the next speaker] I see more than one slide here, so we have a potential problem.

PAPADOPOULOS: Well, I think all the stuff we're doing is like arguing constant factors, so I have "one" slide.

You know, I feel like in the dataflow community that we started out as these outrageous liberals, like we were Ralph Nadar, where any computer out there was unsafe at any speed, and waking up the next morning, I feel like I'm Ronald Reagan arguing for capital gains tax cuts for Intel and Motorola.

So here [referring to a slide] is to clarify *T [20], that we actually *are* talking about long threads in a node. And a specific aspect of *T is pessimistic, and I think this is one of the central points of debate here. We'll issue from a thread many remote loads (and Arvind was calling these "little threads," and I think maybe in an academic sense that's right, but don't think about it that way). It may go out to memory, then be reordered in response; we accumulate these in a join, and when all the responses come back then we launch the thread again.

So this is sort of like a data-driven prefetch, and locality here is locality in your cache. In other words, what's nice is when you start this thread, you're guaranteed all the data there is local.

So that's the pessimistic aspect. If you have enough parallelism, that actually is a very *very* efficient way to do it. But *T really is a short-term exploitation of current technology. And a *lot* of the design decisions here are simply to get out of the chute, moving, because it takes thirty to fifty million dollars and a hundred engineers to design a competitive processor.

But here are the things that I would change. Here is, I think, the way that the community — where *I* see things going (and we're talking about a lot of this). In the threading model, I think it's latency optimization. It's being optimistic. It's caching, it's speculating on cache and synchronization. I think that Tera [3], the von Neumann/dataflow people do it. Sparcle [2], MASA [12], all these people are basically doing that.

Now I think that *is* an important component. I can't do that now, because I can't spend the thirty to fifty million dollars to do that. There is true speculation; I refer you to Tom Knight's work, on Liquid [15]. Also, high-speed state management is very important. And I think there are competing views of replicating register sets, of doing automatic management (and I think the Named State Processor [22] is a very clever view of that). We're just being brute-force in providing a lot of on-chip register to on-chip memory bandwidth.

Finally, I think that you really have to get to the point where you have this total integration of the network layer, that high bandwidth is very important. I think there's a strong correlation here with the generality of the machine. And something not to be missed (which we've all buried) is the scheduling hierarchy, and I think some of the work going on at Berkeley[1] is quite relevant there. We're going to have to understand that, and how that really translates into what kind of mechanisms we use.

IANNUCCI: I think Greg redeemed himself there because he was actually under time, so we will not penalize him for the two-slide error.

GUPTA: OK, I have two slides too, but I claim very coarse-grain writing.

IANNUCCI: This is an optical-density tax-credit that he's claiming there!

GUPTA: So the points that I'm going to argue are more from that of an engineer. I think that multithreaded computers make good sense, but they're part of a whole spectrum of engineering decisions that you want to make to build cost-efficient machines, and they're not a panacea.

So the first point is that the primary thing you want to focus on is latency *reduction*. That means focus on reducing the latency through software optimizations, rather than focusing on how can I throw in bandwidth, how can I throw in multithreading, to *tolerate* the latency. Because if for every operation you're doing, you need compute cycles to hide the latency of the operation that you're doing, and if there are no compute cycles available, then that

[1] The Threaded Abstract Machine [7] and Active Messages [24].

doesn't work,

Obviously, you have to hide the latency in some situations, but there are many different *ways* to hide latency. You can use multiple-context processors, which is one way. You can use weak consistency models (and Arvind had mentioned that). You can use prefetching (software-controlled or hardware-controlled prefetching). You can use dynamically-scheduled processors, which do out-of-order execution and can have multiple references outstanding.

So there's a whole slew of techniques that can be used, and the question is, what is the best engineering? Given a certain set of applications that you want to address, what is the right thing to do? If my goal is primarily data-parallel applications, I will use one set of design decisions. If my goal is general-purpose computing, if I'm building compute servers and doing multiprogramming, I might make slightly different choices.

I agree with Dave Culler that the critical feature *is* the interface to the network, when you're trying to use multithreaded processors. Now, the next point which is sort of related to that is that regular multiprocessors like DASH [18] and Alewife [1] already exploit fine-grain multithreading in many different ways. This point comes out (Anant brought it out too) when I do a write operation. The write just goes into the write buffer and I'm continuing on, executing from the cache; something is going out into the memory system, and the synchronization is happening through the cache. When I issue a prefetch, again I can have lots and lots of fine-grain threads that are hammering out. So to say that just because you have four contexts in hardware, therefore you're only four-way coarse multithreaded, is a totally wrong point of view.

Of course, there still exist situations where multithreading is useful, but when you use all of these other techniques (such as latency reduction and prefetching), you don't need to do this more complex thread-switching too often. And if you're not doing it too often, you don't need to be too efficient. If you take four clock cycles to do a context switch instead of a single cycle, it is OK to do that. The number of contexts that you need, and the amount of state that you have to replicate, is much smaller.

The critical issues, then, are not whether to multithread or not to multithread. Maybe the first thing that we should be spending a lot of effort on is more software-based locality, in terms of compilers, programming languages, and algorithms that exhibit more locality.

The second point I want to stress is the need for high-bandwidth memory systems, where *from* your local processor you get a whole array. If you're trying to do an iterated sparse solver, where you have to stream through a lot of data, then providing enough memory bandwidth is far more important than saying "how do I cut my context-switch cycle time to a single cycle?"

And lastly, I think maybe this is the most important point. As Greg said, people spend a hundred million dollars designing a new processor, and most of these things, whether we like it or not, are going to be used in workstations or small-scale multiprocessors or things that Silicon Graphics builds. Unless the multiple-context processors that we design are also useful in the context of uniprocessors and small-scale workstations, the Motorolas and the other people are *not* going to be designing these kinds of processors. I think that Sparcle [2] or the kinds of things that we are proposing in all our papers are a lot more feasible for small-scale multiprocessors too.

Unless we do that, fine-grain multithreaded processors might always be a technology of the future. Tomorrow's technology, not today's. Five years down the road, they will still be tomorrow's technology. So these are the points I wanted to make.

IANNUCCI: Thank you. I think that that last point is a very good one. Levity aside, I think that's one of the things that's brought a lot of us here, that we want to make sure that that's a "today" technology, that it's not always tomorrow's technology.

AGARWAL: I already spoke for an hour or something like that about twenty minutes ago. I won't say anything technical. Instead, what I'll do is brainstorm about this idea.

So here's a different view of what all this is. So think of dataflow, you know... goodness, it's like borrowing money. RISC's, on the other hand, are like cash now. So where is Sparcle [2], then? Sparcle is a well-balanced financial strategy.

IANNUCCI: Burton, you *must* say something.

SMITH: I have zero slides.

What I'd like to say, I guess, is my perspective of why one wants a lot of contexts, and *when* one wants a lot of contexts. It seems the only reason you *need* a lot of context is when your clock rate is fast. That's because the Japanese memories have a certain latency, and caches have a certain latency, and everyone here knows, of course, that concurrency equals latency times bandwidth. So the amount of concurrency required to handle a certain amount of latency at a certain bandwidth is prescribed. It isn't anything you can deal with. The only thing that's unclear is what is your clock rate? So if you have a very high clock rate and high latency, you need a fair amount of parallelism to deal with it, whereas if your clock rate is relatively low, you don't.

So why *use* a fast clock? I mean, what's the deal? Well, on the network, in particular, it's very important to use a fast clock, even if your bandwidth requirements are modest, because it increases the pin bandwidth, thereby making the copper (or whatever you're using) much better utilized both on-chip and off-chip. But the processor itself might *not* be fast.

It seems to me there's a whole spectrum of design choices, all of which make sense. And I think CMOS processors with 4 contexts make as much sense as glow-in-the-dark processors with 128 contexts. It's just a question of economics. So let's not have any religious wars about that, unless you really think there's something more to it than economics. I don't see it as any more than that.

IANNUCCI: OK, I'll open it up to questions in a few minutes, but I've prepared a few choice questions (panelmaster's privilege).

Well, first let's start off with an easy one. The first question is: how does an architecture qualify for the label "multithreaded"? Go on, Burton.

SMITH: Hardware support for multiple program counters. Not hardware-*implemented*, by the way; hardware *support* for it.

IANNUCCI: OK, care to elaborate on hardware support as opposed to hardware —

SMITH: No, I want to keep it very catholic. Everybody at *this* table qualifies.

IANNUCCI: Want to challenge that?

DALLY: I think you have to go beyond the program counter, and have replicated hardware state of a certain amount of fast storage, typically the entire register set. I think that would be a definition. And the ability to switch rapidly between them. And then the question of what is "rapid" becomes an issue. I think you need those two things: all of the register state, and the rapid switching.

AGARWAL: I think as Burton pointed out in his talk earlier, given that memory systems have a higher latency, you want to exploit parallelism to hide that latency. And you can do it by pipelining, or with what *we've* been calling multithreading, I think.

To highlight the difference between pipelining and multithreading (it's important to distinguish between them): in a pipeline, you are conventionally using a thread where the internal dependencies have been taken care of by the compiler, where you're not splitting it in hardware or at run-time, whereas with multiple contexts, you are taking individual entities that are used from the point of view of a scheduler or run-time system for dynamically scheduling the processor, where the order could be different at run-time than what you might think at compile-time.

GUPTA: I would actually disagree. I think Burton made the key point, which is multiple program counters (and what Dally did was just elaborate on that).

PAPADOPOULOS: I actually would like to raise the stakes a bit. I think there *is* a difference between multiplexing state (and I think multiple contexts in some sense are doing that multiplexing). I'd rather start with trying to figure out what "single-threaded" is, and "multithread" is *lots* of that, or *not* that. And so, if you take "single-threaded" as being that there is one flow of control that you're going to reason about, a processor at a time, then you might think that "multithreaded" is that there are multiple flows of control which you can reason about. And, in fact, if you go do multiplexing, like in switching between lots of register sets (a few, 4, or 128), then it's kind of begging the issue because you're saying "well, is that one processor or am I sort of time-division-multiplexing a processor?"

I think there's the other view, which is much more of a software view, that your reasoning about the evolution of the state of the computation is not *bound* to processors, that you have, in some sense, abstracted or virtualized the flow of control. And you can do that in software. I mean, the Berkeley stuff of having an abstract machine which is multithreaded [7], and then mapping it down onto the hardware mechanism, is a valuable way to treat that.

So what it means to me is support to do that efficiently.

IANNUCCI: So does running Berkeley software on stock hardware qualify the stock hardware to be called multithreaded? The question is about architecture, and we can debate what —

PAPADOPOULOS: Well, I don't know how you —

AGARWAL: Compilation is architecture!

IANNUCCI: ...and engineering and marketing and so forth...

PAPADOPOULOS: I don't know how important this is, because there really are two things. We can look right at the architect's level and we can say "how am I going to do economics of memory, etc.?" But the other one is "what kind of programming models am I going to support?"

DALLY: I think what it boils down to is: if you throw away the state, which is simply an implementation issue, and you said how fast can we switch from one process to another, and we didn't need some abstract measure of where you cross the gray line between being single-threaded or multithreaded, maybe it's one instruction-time, maybe it's ten instruction-times. I think there could be a lot of debate on this table which of those it is. I think everybody here would agree it's not a hundred instruction-times.

IANNUCCI: And so we've already gotten pretty well into question 2, which is: is specific hardware support mandatory? Identify, define and explain. Can you sum up your statement in those terms?

DALLY: Well, I think special hardware support *is* necessary. The best way of doing that is by contrasting with what's available in stock hardware, which is *not* one-cycle switching between processes, and not even ten, but typically many tens and even a hundred (if you have a fair amount of register state). And because of that, you need some way of managing multiple contexts in the fast store of the machine, and being able to make hardware decisions of when to switch. And that requires hardware.

IANNUCCI: Thorsten had a question.

VON EICKEN: Can you explain why the Berkeley TAM [7] model doesn't qualify for "under ten instructions," since it switches in three?

DALLY: You're not changing a complete process state. You're doing a jump, and you're computing, over time, where you want your thread of control to go. But a *process* is a thread of control *and* an addressing environment, and the harder part of switching a process is changing the addressing environment, not changing the thread of control. That's why I disagree with Burton; he thinks that just a thread of control is adequate.

AUDIENCE MEMBER: The difference between you and him is there are different levels of addressing environments. He isn't changing addressing environments at the level of registers; he's not changing at the level of frames, he's not changing at the level of overall address space, so which addressing environment —

SMITH: If I have one big register set, and everybody shares those registers, and the rest of the context-switching is trivial — OK, so now what? That's why I think being too strong about how many register sets or whatever is a little tough. Another way to say it is, I mean, you can sort of say, "...in a number of instruction-times that's far less than the number of registers available in your thread," etc.

AUDIENCE MEMBER: Like asymptotically less...

SMITH: Those are all finite numbers, which are the same. If I can switch every ten instructions, and I only have one register, that's not so exciting.

PAPADOPOULOS: Well, Monsoon [6] only has one register!

SMITH: And it switches faster than every ten instructions.

PAPADOPOULOS: Yeah, that's true.

AUDIENCE MEMBER: Just barely...

PAPADOPOULOS: Just barely...

AGARWAL: How about a slightly different view of that? Instead of focusing on multiple PC's, focus on multithreading, where each outstanding request can be associated with a distinct and unique thread of control. Whereas pipelining, and prefetching, and so on, associate a bunch of outstanding requests with a single thread of control.

PAPADOPOULOS: I don't think — no, that's not good. The Tera machine [3] is a counterexample of that. You can have eight, right? Or if you look at *T [20] —

SMITH: Is that multithreaded?

AGARWAL: Oh, that's a combination of a multithreaded machine and a highly pipelined machine. Yes, you can have hybrid machines, of course.

GUPTA: But I think it also has to do with what kind of software granularity you're getting the parallelism from. What multiple program counters let you do is reason at different levels. Because you have different threads of control that are flowing into the compiler, for example, in the pipelining model, and yet within a single flow of control, you think of a reference that is five hundred away from you, and to reason with that becomes very very difficult.

Totally separate them out, and exploit a combination of both. What the pipelining, prefetching and things, are letting you do is exploit very fine-grain parallelism in the program. And what the multiple contexts are letting you do is different loop iterations, different tasks. They're totally separated from each other, and you hide different kinds of latency.

PAPADOPOULOS: I think that's one model. I agree with your coming back to a programming model, and I think that's essential. We're arguing at where the continuum is.

SMITH: In fact, multithreading is a special case of a more general idea that dispenses with the program counter notion at all. It says that what a multi-*context* machine is, perhaps generally, is something that provides the same programming model among and within processors. That is, it exploits the parallelism in time in the same way it exploits parallelism in space (thereby simplifying the compiler-writer's job).

PAPADOPOULOS: Sounds like dataflow.

IANNUCCI: Sure does to me!

SMITH: Dataflow machines and multithreaded machines (in the sense of multiple program counters) are both special cases. But you're also assuming MIMD, in this model. SIMD in time and space both, for example, that's something else; I don't know what you'd call that.

GUPTA: In traditional dataflow, the main thing is the fine grain of synchronization, where you have explicit instructions that spawn off continuation threads for each tiny operation, rather than synchronizing everything at a coarse-grain level. In the Jade [17] programming language what you have happening is you have tasks, and then in the main procedure you specify what matrix goes where, and what tasks you're going to do. That's dataflow, right? Except that your dataflow data objects are just very coarse.

PAPADOPOULOS: Right! So it's all a matter of partial versus total ordering, right? And if you're talking about the entire program, every operation is partially ordered with respect to other ones, and that's dataflow. If you start imposing threading, then you're starting to impose more and more ordering constraints. And where we're talking about in this continuum is, well, what are the sizes of the things that I'm going to total-order? Are they on the order of one instruction, ten instructions, a hundred, a thousand instructions? How does that interact with your ability to discover those regions?

GUPTA: I think a lot has to do with applications. If you talk about a class of machines where you're going to be using a thousand processors, what are the applications which are going to need a thousand one-gigaflop processors? What is the structure of those applications going to look like? Those are not going to be my Emacs and regular compilers. And if the majority of applications that are going to be using such machines are going to be data-parallel applications, or dealing with large matrices and things like that, then from a cost-effectiveness point of view, it might just make sense to do a lot more coarse-grain operations rather than fine-grain operations. You need to bring the application into the picture. You can't leave the applications out, and then talk about engineering decisions.

DALLY: This is one of these self-fulfilling prophesies (I think Arvind said it earlier), that if you make exploiting fine-grain parallelism very expensive, nobody's going to use it, and all will say "well, the only way to solve these problems is to do big data-parallel programs."

There are some programs that work well that way. But there are styles of programming and problems to be solved which would make very good use of fine-grain parallelism, and the point is that it's not very expensive to implement something which will switch at a very fine grain *for* that fine-grain parallelism. I happen to have it in my pocket here: a multithreaded processor[2]. And you can build, adding very little logic to a standard processor, the means to do this.

[2]At this point, he pulled a Message-Driven Processor [8] out of his pocket.

So it's not a question of *what* you're giving up for it. It's a question of "is it desirable?" and the answer is yes.

PAPADOPOULOS: I don't know whether I want to jump on Bill first or — Let me just answer something. I gotta get a jab in. My jab is, Anant, I mean Anoop, — essentially this is directed at both of them — you ought to get out of the cache business, because the data-parallel machines now, there's no data-parallel machine which has cache in it. You categorically don't need —

GUPTA: CM-5 doesn't have a cache?

PAPADOPOULOS: Oh boy. You know that it doesn't cache global data and references, OK? It's not a *coherent* cache. So you might as well not do that because these data-parallel codes *clearly* don't need anything, *clearly* don't need the cost in extreme complexity you add —

AGARWAL: You're wrong, Greg. You're just totally wrong.

PAPADOPOULOS: I'm not wrong! Go out and show me a machine which is a data-parallel machine, which has a globally cache-coherent memory system in it, OK? It's not there. You don't need it.

AGARWAL: So all you think is all the machines out there are the best machines in the world and the kinds of machines you should be building?

PAPADOPOULOS: Show me a class of applications in which, you know...

IANNUCCI: OK, I want to go on to some more questions. You'll get a chance to take your swipe back. Just wait a minute. Let's start...

AUDIENCE MEMBER: The word "cache" pops up from time to time, or rather often, Could you define what we know caches *can* do, and what we know caches *can't* do, and what we don't really know, which is in the wish gray zone?

IANNUCCI: That's brings up a very good question. Let me just skip out of order here so that we can focus on a concrete question. What role can or should caches play in multithreaded architecture? Identify, define and explain. Can we go down the line here?

AGARWAL: Let's make two things very clear. To answer your question, first, the issue of locality management, and second, mechanisms to do it. I think from *my* point of view, from my, call it my religion, or call it an axiom upon which I base all my research, is that if you want to build large-scale parallel machines, if you want to be able to run massively-parallel computations on massively-parallel machines (really, in two-dimensional space, not in virtual space) then we simply *have* to understand how to exploit locality, we have to understand how to have computations running which don't run into communication latency. We *have* to understand that. OK, that's my view as an axiom.

But the next step is, how do we do that? Clearly, our group is working and many other groups are working a lot on compiler and run-time techniques, software techniques for locality management, and the state of the art is such that we don't completely understand it. We're learning a lot! What you can do now and what you could two years ago, there's been a quantum jump. But there's still a lot more to do.

Caches: I think caches are a phenomenon of, let me put a year on it, 1995. (I shouldn't say "caches," I should say "hardware-managed caches," because "cache" is really a local fast memory, and that's going to stay forever and ever and ever.) But I think the issue at stake is hardware support for cache coherence, and I think *that* is going to be there till 1995. And if you look at the limitless coherence scheme we support on the Alewife machine [1], just *some* of the common cache operations are done in hardware support for cache coherence, and all the other cases are handled in traps and software. And we've got some simple back-of-the-envelope calculations that show that in 1995, speeds of networks in relation to processor speed will be such that *complete* software handling for coherence, *dynamic* software handling for coherence, will be possible.

So I think this is sort of a long answer to your question, that it's a mechanism. I think mechanisms will change as we understand more about what we can do in software.

IANNUCCI: OK, Bill requested that we go down the line and give each panelist a chance to respond. I'd like to do that.

DALLY: Let me resonate with what Anant said, in some ways, in that, I think what's *essential* about a memory system for multithreaded processors is that it do two things. One is that it be able to do remapping to get the advantages of memory utilization that I referred to in my position slide, in that, to do that, you have to be able to take the storage at all levels in the memory hierarchy (the register file, the first-level cache, the second-level if you've made the mistake of putting one of those in, the main memory) and take that storage and allocate it among the threads in a way that when one of them doesn't need a lot of storage, another one can use it. Because that's the big gain here. The memory is the expensive part; the processors are cheap.

The next question is, once you have this mapping facility, how do you manage it? And this is where I think you get into an issue where, once people start referring to it as their religious beliefs or an axiom (which is maybe a euphemism for belief), it means they don't understand it. If people really don't understand how to improve locality in these machines, and the question of how automatic you can make it is an open one, then what mechanisms you want for repositioning data, to try to reduce the latency in certain cases, I think is an open issue. But I don't think you should forestall the possibility of doing it by not putting in the ability.

In terms of hardwired mechanisms, I think today you don't need them. If you look at the ability of how long it takes to get a word from remote memory in these machines (120 cycles, 50 cycles in some machines), the little you gain by hardwiring in a coherence protocol (which might be ten cycles out of a hundred) isn't justified by the flexibility you give up in locking out alternative protocols.

So I think even today it doesn't make sense to do your cache coherency protocol in hardware. You should build a message-driven processor and have it interpret the protocol.

PAPADOPOULOS: Look, there's no argument. Anybody who argues that you shouldn't look for locality is off on a different planet. I mean, everybody knows that that's important, that's why we *have* register sets, that's why we *have* activation frames. There clearly is a storage hierarchy in machines, and it *does* come down to what is the automatic support that you'd like for that?

So, let me disabuse people, though, in the audience (my friends here won't) to the relative *complexity* of coherent caches. That, I think rightly so, and that isn't their duty, can be viewed as a layering on top of a pure message protocol. I think that pure message protocols should be exposed to the processor directly, and exposed as efficiently as you can (that's a common case, guys), and that these cache coherent protocols are *exceedingly* complex. I think that they teeter on the edge of things that you would think is rational to put in hardware.

So I think what it comes down to, I don't believe in coherent caches the way they're being presented now as magic hardware in a way, because I think compiling analysis can go a long way. *The* key point (I think Anant mentioned it) is what do I really need in terms of a trap (or something) to help me get out of the problem of these coherence problems, of trying to maintain their consistency in some way? And do I really need to start doing software caching and speculating on these? Do I need some sort of help, kind of like in the way I rely on virtual memory to check for page faults? I don't know. But it *certainly* is not the kind of complexity proposed in machines now; that's completely false.

IANNUCCI: Anoop?

GUPTA: OK, a couple of things. Firstly, I think that people who *are* building cache-coherent machines, we're not saying at all that the basic interface to the network and the messaging should not be supported. I think that should certainly be made visible to the program level (that means to the compiler writers and the software engineers). I think what we will see is more and more of the things that are handled in hardware for cache coherence migrating into the software.

But to say that *all* of it will totally be done in software, is highly unlikely, because every reference that comes in from remote, you're going to interrupt the processor, take whatever number of cycles (five or six or four) and do the context switch. You can build very specialized processors that can check certain things for you, the common cases, so you want some minimal amount of support in hardware for providing cache coherence.

A final point regarding whether cache coherence will eventually stay around or not, (which I guess we will see as the years come): What is the software model that you're trying to address? In my opinion, the majority of machines are not going to be the kind found in national labs, where every national lab has a small number of them. I would like every research group, in addition to their workstations, to have a machine which might have 128 or 64 processors with several gigabytes of memory. On this machine, I want users to run serial programs, or programs with just small-scale parallelism with 4 processors, 8 processors, and have everything in this relatively simple model. I want multiprogramming, regular operating systems, editors, everything else running on the same machine. If I'm doing all of those complex things, and you're going to handle all of the software cache coherencies in hardware, I think it's going to be very difficult.

So there are lots of complex tradeoffs that arise when you think of how these machines are actually going to be used (or at least how the kinds of machines that *I'm* hoping for are going to be used), and whether cache coherence is going to make sense.

Exactly what does get done in hardware, and where the software-hardware boundary is, is probably going to change.

PAPADOPOULOS: I ought to disagree with that. It's just that right now it is not a panacea.

IANNUCCI: OK, I'd like Burton to answer his question.

SMITH: I have a couple of things to say about caches. I have to say that these opinions are my own and not those of Tera Computer Company. There are people in the audience who may not agree with this. In particular, David Callahan doesn't like caches one little bit. So let me just say something nice about caches, I guess, but not too nice.

The first item is that cache lines, as usually put together, are basically unit-stride vectors. They are a pain in the neck to the compiler writer. They're a horrible thing to deal with. The reason they're there at all is because the line length, the very fact that there are multiple words in a cache line, is a source of parallelism. It's a source of parallelism that is used to help the cache overcome the memory latency *it* sees in trying to amortize the cost of a line fetch by fetching more that one word. Now in an architecture in which we have other ways of trading parallelism for latency, we probably don't need multiword cache lines. If we didn't have multiword cache lines, then our compilers would likely have a much easier job regardless of what other decisions we make. So I think for multithreaded architectures, cache lines should be one word long (if they exist at all).

Item 2: coherence is necessary when dependences are unknown, and not really too necessary when they're all known. So I think it may depend on what your language is. If you have a language which tells you where all the dependences are, then I think that you should throw hardware cache coherence (in the usual sense) away. And if you have a language in which it's *not* known, then you need to do something.

Again, my favorite idea is not to have these surprise invalidations, but instead to try to figure out a way to control when to cache. And this has to be done in a rather sophisticated way. A lot of fairly simple attempts to do this have not been uniformly successful. Basically, the way programs behave, variables change state from being cacheable, that is, being essentially constant, to not being constant. And *when* these phase changes occur is complex and takes a lot of analysis. But I think that's a research direction I guess I'd like to see pursued.

How to design hardware, perhaps address bits or something, I don't know what, that allows us to change the cacheability state of variables, as part of the reference, without having too much management of the bits that we use to do that, is a hard problem, I think.

But anyway, that way you'd only have invalidation of the cacheability state change from cacheable to uncacheable. We'd have *no* invalidation traffic whatsoever, compared to what we have now. I think that's hard to do, but I think that's a direction I'd like to see pursued.

IANNUCCI: I'd like to switch gears for just one second. Let's do a lighthearted question. Discuss the following in the context (pardon the pun) of multithreaded: shared memory is to message passing as walking to school is to carrying one's lunch.

We should have the Jeopardy music! [Starts humming it.] This is from Burton. ... Any comments?

SMITH: For 500 bucks?

IANNUCCI: "Multithreading" for 500, yes!

SMITH: Yeah, I think in the context of multithreading, shared memory and message passing are perfectly compatible with each other, uniformly desirable, and even if you have to *buy* your lunch, it's OK.

IANNUCCI: Do we have consensus on the panel?

DALLY: No, I think you have two things up here which are incomparable, because shared memory is a programming model, I mean a shared address space where everybody refers to the same address space on their nodes, whereas message passing in general is a hardware —

IANNUCCI: But this is a widely misunderstood thing. So please, continue to dispel.

DALLY: Message passing is, in general, a hardware mechanism that you use to *implement* shared memory, and you may also use to implement other models of programming, like dataflow, or a message-based model, but message passing is a hardware substrate on top of which you build things. Shared memory is a programming model. You can't compare them as alternatives. They sit above one another in a structure.

I think that one thing that's important in looking at multithreaded processors or any other architectural feature is to separate three things. One is what the hardware is, the machine, and that's things like what topology you chose for your network, how many registers you have, how fast is your ALU. The next is what your mechanisms are, like you have a message send, and you dispatch processes for message arrival. And the third is what programming model you run on top of it. If you choose your mechanisms right, you can run any programming model you *want* on top of it. So you can have lunch, dinner, whatever.

IANNUCCI: Greg?

PAPADOPOULOS: Well, I have to respond to Bill. I think this has been clearly a point of disagreement. That dataflow is not a programming model —

AGARWAL: It's a religion!

PAPADOPOULOS: A dataflow machine clearly articulates the mechanism for interaction as passing tokens, i.e., message passing. So I don't want to hear that if I build a dataflow machine, I've actually built one of your machines. So I might even turn that around.

I think the other end of this is that there is a message-passing programming which is pursued by people who do, well, object-oriented is generally on top of that, but you can clearly find people who program machines using message-passing programming models. And the shared memory, to me, is really just an optimization that's important, because we want to be able to freely reference objects in our language on a heap...

GUPTA: Message-passing is the base model that everything is going to be built on, I mean, I don't know if you can say —

SMITH: [Sarcastically] But *shared memory* is what it's built on; message passing is a veneer on top of *that*!

GUPTA: Exactly. I'm saying that the level is right; if you're sending data over the network, the question is, what do you implement in hardware in very simple finite-state machines, and what gets handled by a general-purpose processor? One view of message passing is it goes and interrupts the general-purpose processor and the receiver does something. In a shared memory, there might be a small memory system controller, a sophisticated memory system controller, a programmable one that takes a thing, looks it up and sends it off. So in addition to the data processor, you have a synchronization processor and a read controller, and you might have a variety of small finite-state controllers, and then you would pay less overhead than if you did everything in the general processor.

So it's really a tradeoff of an engineering issue. What is the common case, what do you optimize for, rather than saying the basic thing is message passing, and shared memory or single address space is on top of that, or the basic thing is single address space with message passing is on top of that. Either of them will make sense.

AGARWAL: Let's see, I've been stacking, on my little stack here, things to respond to.

IANNUCCI: Well, start with the question, OK?

AGARWAL: Now that I have the microphone here...

Very quickly: Greg made a comment a little earlier about complexity, where he said he wanted to disabuse the audience here that these are very complex, so let's do away with them. I *do* agree that they're very complex, and we should, at *some* point, try to understand them. Because really, cache coherence protocols are really mechanisms for replicating data and ways of managing them. Whether you implement them in hardware or in software, you don't have to understand the complexities of the interactions and so on. I think it's wrong to say, "it's hard to do, so let's walk away, even though it gives us performance." That's like saying "processors are hard to build, so let's build a one-bit processor," or something like that.

I think as architects, part of the job is to understand complexity. Once you understand it, it becomes standard practice. So that's one comment on what Greg said.

In fact, on shared memory versus message passing, I think I agree with about a third of what Bill said. Shared memory, message passing, many of these things that we call machines are really programming models. Pure functional programming is one class of programming styles. The Fortran/C view, the shared-memory style of programming, object-oriented styles, there are a whole bunch of styles of programming. And in the past, people hardwired machines to implement one such style. Almost like Fortran engines or almost like object-oriented engines.

I think what's happening now is that we're very clearly understanding that architecture is really engineering. And here are all these programming models that people like to write programs with. And I think as engineers, or architects, our goal is, what is the right hardware to build, given that we want to target this machine for the technology of 1997? And understand the technology at that time, and implement the hardware that will run these things efficiently for that point in time.

So I think this is really an issue of engineering.

IANNUCCI: Yes, Bill?

DALLY: OK, I just have to respond to a couple of comments I think Greg and a couple of other people have made, for they're making some gross misstatements of fact, because they're confusing models with mechanisms. And I think message passing has a tendency to do this because there *are* message-passing programming models. The synchronous ones, like CSP [13] or Occam [19] languages, are one example of that. There are also asynchronous ones, like ones on which various actor and object-oriented languages are based. Those are very interesting programming models. Their underlying message-passing *mechanism* is quite a bit different than that. And if you look at a programming model (we'll take one of the actor ones, for example), there's a lot of semantics wrapped up in what a message is in an actor programming model. It involves naming systems, it involves synchronization, it involves lots of process scheduling and interaction,

What you do when you define mechanisms is you look at a lot of different programming models (the actor model might be one; shared memory might be another), and you strip out the naming, and that goes into a naming mechanism. You strip out the communication, and that goes into a message-passing mechanism. And it just so happens that no matter what you're doing in these machines, you have a network, you gotta get some bits from here to there, and that's sending a message. And that's why the underlying message passing is what *all* these models wind up getting built out of.

Now some people confuse shared memory the same way because they've taken their programming models for years, and have been building one-model machines by wiring the model in, and therefore calling it a shared-memory machine. What that really means is that they just didn't design their mechanisms to be general. They came up with a set of mechanisms that could handle only one model.

I think when you talk about dataflow not being a programming model, that's where the confusion arises, because people have designed mechanisms which were skewed very much in that direction, and aren't as good at implementing other models.

IANNUCCI: OK, related to that, I'd like to ask the following question: to what extent *should* hardware mechanisms, if any, be oriented toward a particular operating system's threading model? Identify, define or explain. And if so, are there any standard opportunities here? Burton?

SMITH: A particular operating system's threading model, in my opinion, is a brain-dead notion. I don't think operating systems should *have* threading models. I think all the operating systems I know of are handing those threading models over to the user as quickly as they can. So I would reinterpret this as meaning "operating system vendor's user-level threading model," perhaps. Or something. I'm not sure how to put this, but I don't see threading models as staying privileged for very long.

So I don't think you really need to do much of anything. I think the question is just not too significant. I think you can support most anybody's threading model on the kinds of computers that we like, without very much hardware mechanism over and above multi-threading (*however* we collectively define it).

IANNUCCI: Is there a consensus?

AGARWAL: I wish I could say I'd start a company, and go out and build a machine, and make lots of money. If you imagine that there's this neat language or neat operating system that's going to rule the world, then it probably behooves you to hardwire that mechanism so that it's very very fast. Look at modern RISC processors: the only things to distinguish them are small little mechanisms somewhere that makes a big difference to some language, like when you do byte addressing and so on to make string ops work well and things like that.

But I think from a short-term point of view, it makes a lot of sense to hardwire specialized mechanisms. But probably not from a long-term point of view.

IANNUCCI: Arvind has a question.

ARVIND: I think that we are dwelling too much on the internal details of these machines. I have a question, because we keep alluding to the common case, and I don't know where these common cases come from. What I would like to find out is (because everybody here is building a machine and has a software system that goes along with it) do you think any significant changes in your hardware or software is needed, in order to run the compiler itself, on say a 16-processor version of your machine, or a 4-processor version of your machine, or a 100-processor version of your machine?

SMITH: The answer in our case is no.

AGARWAL: The compiler is another application, and I think if you're feeling —

ARVIND: That's how I'm feeling. I want this to be part of the real world.

AGARWAL: Let me give you one example. If you take as an application, the problem of cell placement on VLSI chips, it's really a packing problem, where you have a massive area with a bunch of blocks, you want to place them and minimize wire lengths and so on and so forth. It's an application. We have parallel programs to do that efficiently.

Now, the placement problem for compilers is exactly that. The placement problem for compilers in large machines is: given a bunch of threads and data on nodes with communication links, how do you do a good placement of these threads in the machine to make them run very fast? That's a *basic* problem.

So clearly, if you have a compiler to do that, then your compiler is no different from the other applications, just a different viewpoint. So your parallel compiler —

ARVIND: You're running all over the place, OK? Do you *have* a compiler for your machine?

AGARWAL: Yes.

ARVIND: Can you run it on your machine?

AGARWAL: We *will* run it on the machine, yes.

ARVIND: All right, so you're saying your placement techniques are sophisticated enough today.

AGARWAL: No, what I'm saying —

ARVIND: You can take a program of the complexity of the compiler and distribute it over —

AGARWAL: No, what I'm saying is that the kinds of problems that compilers have to solve are very similar to some of the applications that they have to compile. So that's a compiler —

ARVIND: I take that for granted. It's a question of complexity.

AGARWAL: For many phases of compilers, they have to be able to run efficiently as parallel programs. In fact, if you look at Geoffrey Fox's work [4], a large part of his actual computations are remapping of data, which is actually dynamic compilation. So yes, I firmly believe that compilers have to run in parallel. And our machines certainly do that.

ARVIND: That's *not* the question I'm asking. I *want* to run the compiler on your machine. The question I'm asking is: do your techniques, software or hardware, require any enhancements to be able to deal with that level of complexity?

AGARWAL: The same architecture will give you certain phases of compilers running in parallel. We probably won't have to do anything special because the compilation itself is another application.

ARVIND: So why don't you start producing numbers on your compiler and these things instead of matrix multiply and eigenvalue problems?

AGARWAL: Well, if you want to quibble with me like that, you haven't done it either. I'll do it eventually.

ARVIND: You mean that I don't have a compiler written in Id [21], all right? What I'm asking is, you're writing a program in Fortran and C, and *you* claim that your fantastic partitioning techniques... What I want to know is, are the hardware and software at the level right now which can take *your* own written programs and run them on these machines —

AGARWAL: I'm not claiming that I understand locality management. I'm saying our research project is focused on trying to understand these issues. Now, one or two or three years from now, if we succeed, there will be a partitioning compiler, and a placement compiler, and clearly, we will throw the compiler itself at itself. Compiling an application. Now if we succeed, then our work will be solved. But if we fail, then I'll throw up my hands in three years and say, "we have to have programmer annotations for locality management." So if these things are very important we can do it. But today I don't understand how to do it.

PAPADOPOULOS: I'd really like to go back to your question. I think it was pretty good.

ARVIND: But I want to hear answers from Stanford on this.

GUPTA: Hold on for a second. Let's go into compilers. A current compiler processing at 25 MIPS, 100 MIPS, 500 MIPS, doesn't take much time. But compile times are going up because of new optimizations and analyses that are being added. A lot of the ordinary compiler things that you do, if you look at existing compilers, I would rather do in parallel.

When the compiler should need some of the optimization techniques that we're looking at, they start with some of the optimizations that take a very large time. Most of those optimizations turn out to be massive search problems, often having to do with complex register allocation and things like that. We can quite efficiently run these phases in parallel. So people have taken those specific phases and implemented them as parallel algorithms. And that means we understand some of the weaknesses of the architecture that are there, again, if they're there in the prototype, and you have a good model for what it's going to look like. But a lot of those things, yes, you can run in parallel. No problem!

ARVIND: Just that we're not there yet.

GUPTA: We are not *focused* so far on parallelizing compilers. If you take a specific — there are people who have taken the register allocation codes and written them in parallel. Yes, yes, we've done that.

IANNUCCI: Greg, you wanted to have a chance to respond to the question?

PAPADOPOULOS: Yeah, because I think this really gets to the core of it, from my point of view, what the commercial consequences will be in twenty years. It's really not so much multithreading, it's communication. And the current view of communication is something like Mach [5], which is a kernel activity.

And presently, what we're all in total agreement with, in our machines, is that we're after efficient user-mode communication, in that the interprocessor communication should be exposed to the compiler, and you should get at that very cheaply and easily, and I might actually refer to the MDP [8] as a particularly *good* integration of communication to a processor in a model that I certainly would want to emulate, in looking at that. But you're exposing that, you're also exposing scheduling, so all these things like threading, communication, those don't belong as part of the operating system, they belong as part of your reasoning about machine language that your compiler is permitting.

Multithreading is a technique for managing that. It's a technique for managing now the kinds of interactions that you're going to be having that the compiler is seeing. And that is a style of compilation, it's a style of support that you may need to various degrees depending on what the language is that you're starting with, to help you use that. And I really believe that *that*, if we are making a contribution or *will* make a contribution, is to get the numbers that might be seen now on the Intel machines or the CM-5, of 25 microseconds to send a message from one side to another, to something that's on the order of tens of nanoseconds or hundreds of nanoseconds, in terms of overhead. And *that's* the big step. And I think everything else that we're doing here is to reduce overhead.

No, it's not part of the operating system, but the operating system and the compiler for the operating system can use those techniques as well. Nothing I want to sneer at.

SMITH: Just a little quick note. Mach has given up on, I think partially but, *privileged* threads as well.

PAPADOPOULOS: Well, I know. Yeah, I mean, clearly, the threads always have seen their abuses there; I didn't want to...

SMITH: Right out of the kernel...

PAPADOPOULOS: But you don't get much support now.

IANNUCCI: Any last questions before we wrap up? Kevin has a question.

THEOBALD: About Anant's comment equating VLSI placement algorithms with thread scheduling for compilers: would you elaborate on that?

AGARWAL: One of the phases in a compiler (every single compiler *we're* building) is a thread-placement phase, where the compiler looks at all the threads and the communication patterns between them. It tries to optimize the placement of all those threads on the machine. It'll try to place communicating threads close together. And it uses techniques borrowed very heavily from VLSI placement, from Leighton and Leiserson's work on optimal embeddings of cells, and so on. Since we're trying to decide what is an optimal placement, the same techniques that we can use to parallelize those cell placements, we can also use to parallelize *that* phase of the compiler.

IANNUCCI: Well, unless the panel wants to respond, I think we're going to have to wrap up.

3 Panel Session II: Programming, Compilation, and Resource Management Issues for Multithreading

Panel chair:

Robert H. Halstead, Jr. .... DEC Cambridge Research Lab

Panelists:

David Callahan .... Tera Computer
Jack B. Dennis .... Massachusetts Institute of Technology
Rishiyur S. Nikhil .... DEC Cambridge Research Lab
Vivek Sarkar .... IBM Palo Alto

HALSTEAD: The topic of this panel is, to put it very briefly, the software side of the issue. We've talked a lot about multithreaded architectures. David Culler has given us a *little* bit of a transition into thinking about things from the software side. But whether or not you think that software makes the hardware mechanisms redundant, or simply that the existence of various hardware mechanisms creates new challenges on the software side, the purpose of our panel discussion is really to address the question of what else changes, besides the hardware, when you consider multithreaded computers.

I'm very happy to have been able to assemble the following panel for this discussion, and I'd actually like to invite you folks to come up and sit in the hot seat. We have David Callahan, who I think of as one of the chief thinkers (especially on the software side) for Tera Computer; we have Jack Dennis, who has contributed all kinds of things to computer architecture over a very distinguished career, and, of course, was one of the first people to be talking about what we now recognize as multithreaded architectures; we have Nikhil, from the DEC Cambridge Research Lab, who's made a lot of contributions to the *T work we saw this morning, as well as on the software side; and Vivek Sarkar, from IBM, who has done a lot of good work in the area of compilation for parallel machines.

I wasn't quite as swift as Bob Iannucci in saying that we had to have no more than a certain number of square millimeters of ink used by each panelist, and so I haven't specified anything about the number or density or color quality of the slides. However, I *have* set down the rule that I would ask each panelist to take no more than five minutes to give their view on the various issues that are related to the software aspects of multithreading. And so what I did was come up with a list of questions.

The first question, which I have to credit Jack Dennis for, is: what *will* multithreaded processors be used for? We may all be assuming we know what they're going to be used for, but it'd probably be useful for us to know whether all our assumptions are the same or different. So what is the context in which these processors are used?

Where will the parallelism come from that it takes to utilize these processors? Explicitly parallel programming? Compiler-discovered parallelism? Some mixture? David Callahan tells me maybe we don't *need* that much parallelism, or whatever, so he can defend that statement when he gets up here.

CALLAHAN: Assuming I really made it. People put words in my mouth all day!

HALSTEAD: All right, well, that keeps life interesting.

A third question is: what are the various resource management issues that come up with handling these contexts and other finite resources that exist in a variety of these new architectures?

What is the importance of locality of reference, and what can we do to enhance it?

How does compilation or optimization look different when your target is a multithreaded processor?

Are there any new principles that are sort of generic across multithreaded processors, that are emerging from our study of these things?

And finally, something that I don't have any answers to, but I think is an important question: how does the use of these kinds of architectures affect debugging and performance measurement of programs?

So this is the set of issues. We may be able to add other things to them, but what I'd like to do at this point is really sit down and invite the panelists to come up and discuss it. And for lack of a better order I thought I'd use alphabetical, although if anybody is particularly eager to go first, I don't really care that much who comes up here next.

David Callahan:

CALLAHAN: Let me start by talking about how we see (or at least how *I* see) multithreaded architectures in their perspective. Tera is a high-end supercomputer company. Multithreading is a vehicle toward high-end performance goals, more than anything else. I know many people have an agenda to see this kind of architecture replace conventional architectures across all of computing. And I don't really see that happening any time.

However, I *do* see it as a very *effective* vehicle for extending shared-memory programming models into a high-performance arena. Now I suppose Dave Culler might think that that's probably not the best plan, but it certainly is a good plan in terms of getting a machine to market, at least in the short term.

A lot of our thinking, then, about the role of software versus hardware is colored by this bias in this perspective, so if there are differences between me and some of the other speakers, many of them may derive from that.

Now the other thing, of course, is compiler people in the research area have a tendency to be something like Nietzsche's camels. They'll take on all these problems, and say, "sure, sure, we can solve them," when in fact they probably have very little chance for this.

The numerical algorithms people have a notion that I think is very valuable, and maybe we can borrow it for a few moments, and that's the "stability" of an algorithm. An algorithm is stable if small changes in the input lead to small changes in the output. And this is a good question to ask about any particular problem you're casting for the compiler to do. Is it a stable problem? Can you reasonably expect small changes to lead to small changes? We want this because

1. We want to be able to write programs whose behavior we can predict *a priori*. We don't want to have to experiment within our design space to find out what's really going to happen.
2. We want to make sure that if we're going to push some problem off onto the compiler, that it can solve it for nice broad, general classes of issues.

So here are some nice stable compiler problems. I mean, these are things that they've been doing for thirty years, and continue to do well, with incremental improvements: instruction selection, register allocation, (and likely with the advent of RISCs) pipeline or very fine-grain scheduling. They do these things really well. They're nice, localized problems, and compilers do a very good job on it.

Here are some unstable problems (or at least things I *conjecture* to be unstable). Unlike matrices, for which you compute a condition number and you can use that to judge stability of algorithms, for these kinds of problems it's hard to judge, hard to quantify stability. So here are things I think compilers are not going to do a good job at:

- Large-grain scheduling: taking a whole problem, mapping it onto an ensemble of processors. It's clearly intractable.
- Large-scale parallelization. (So small-scale parallelization is vectorization. I mean, compilers are reasonably good at that. I'm not sure I'd call it stable, but they're reasonably good.) They're not going to be good in here, because as you get scale that's larger and larger, the likelihood of hitting one of the little things that confuse compilers becomes very high, and you're going to trip up.
- Large-scale memory management. By this, I mean the distribution of arrays across distributed memory. Compilers are not going to solve this problem (I conjecture) though it's an active area of research. It's just too unlikely for them to be able to have good universal solutions.

So this *implies* that there are some things the hardware and this nebulous runtime need to be able to do. These include dynamic processor management (all the scheduling issue has to be deferred to runtime). You're going to need some form of explicit parallel constructs or language that has a lot of implicit parallelism. And I think you need shared memory, where shared memory here means that the placement of data values in the memory system is not a first-order effect on performance. And these are all things that are handled very well by multithreaded architectures (or at least I *believe* are handled well by multithreaded architectures).

On the other hand, they may be handled well by *any* parallel architecture, assuming you're willing to throw processor utilization away. Dave Culler's slide where he showed that you just take a whole lot of processors, and think of multiplexing as simply condensing them down for efficiency reasons, is, I think, a very good way to view the kind of machine Tera has built [3].

Now there *are* some *special* problems brought on by multithreading that are new and, I think, somewhat unique in this light. One is the use of light, very low-weight synchronization, so that you can now apply synchronization in contexts where it was not profitable, where you'd have to give up on parallelization. And this is one of the things we'll be doing. This is the use of redundant or speculative computation. You now have more hardware resources then you may be able to use, and how can you use those effectively? I've been doing some work in the area of recurrent solvers. They rely heavily on redundant computation. This is something new. Both of these are fairly localized, and therefore stable, issues.

Then there's a question which I call "parallelism packaging," which basically means making decisions about whether to stay on a processor or to go off a processor. It's not entirely clear that you can solve these, if this is a stable problem or not. It's very sensitive to architecture. I think we can make it stable by having mixed dynamic and static solutions.

For dividing, well, it's just another ten thousand–processor machine, so if you can solve that problem, multithreading isn't really any different. Naming, scheduling, control issues, race conditions: nothing new there. It's not any easier, of course.

And then finally, for performance measurement, which was one of the other things, we have the big problem of accounting for sharing. Since we're a shared-resource kind of machine (that was the buzz-tag on the HEP [16] years ago: shared-resource MIMD computing; and Tera's the same), we have this issue of: if you're sharing a processor with somebody else, how do you factor out their effects? If somebody else generates a hot spot in the network, how do you factor that out, or do you just run again and hope they're gone? Do you shield yourself in some subsection of the machine?

And then there are, of course, memory anomalies that must be debugged. I've been lobbying for a feature in the machine, such that if any read takes a long time, like two thousand cycles, regardless of whether it succeeds or fails, the stream that issued it will take a trap, and log it fast, so you can find out what happened. This kind of stuff is going to be very unique to multithreaded architectures.

HALSTEAD: Thank you.

DENNIS: I'd like to just review something to begin with, and that is sources of latency, and the techniques we use to attack them. We're very familiar with memory problems since they've been emphasized many times. We have local memory access, which is to the DRAM that's local in a processor board or chip, and we have remote or global memory accesses, where the access must go over some kind of network. A third source of latency is scalar operations, especially floating-point arithmetic, which typically takes more than a few cycles (or more than one cycle) in a machine.

Several methods of dealing with latency have been introduced in this workshop. One way is to avoid it by using cache, and hope that you get repeated reference to information in memory, so that successive references will be satisfied by the cache. A second technique is context switching: switch to another activity so that we can hide the latency. And a third is interleaving of threads in the processing element itself. This is the technique which has been pioneered in the HEP [16] and used in the Monsoon processor [6].

Now I'm going to point out two domains of applications which seem to be dominating, or somewhat dominating, in the present world of computing. The first one I'll call "parallel scientific computation," and is exemplified by the Tera computer [3] and other attempts to build high-performance supercomputers. The second domain I'm going to talk about is workstations and servers, which is another large potential area of application for high-performance processors.

These seem to me to have very different characteristics. For example, in high-performance scientific computation, we all know that floating-point performance is key. You can't sell a machine unless it has top-notch floating-point performance. Fine-grain communication is becoming important, because if you are stuck with *large*-grain communication, it introduces latencies and buffering problems which reduce the amount of parallelism that you can exploit in a large-scale computation. A deep pipeline is feasible, or even essential, if you put floating-point units in the pipeline of your processor. All three forms of latency are relevant here. In particular, the latency of scalar operations must be addressed to get that high floating-point performance.

All mechanisms are applicable: the cache mechanism, the context-switching mechanism, and interleaving in the pipeline. All mechanisms are relevant; maybe all but the cache is used in the Tera, for example.

The cache mechanism has a different role in a multithreaded computer. If one is hiding latency, or tolerating latency, by means of some other technique (by context switching or by interleaving), then the role of cache in reducing access-time to memory becomes unimportant, and the *important* role of cache is in reducing memory bandwidth. So it's very likely that in multithreaded computers, much smaller caches will be feasible and appropriate.

I expect that implicit parallelism will be the dominant way of programming for these machines. We already see it in the evolution of Fortran D [11] and, of course, the interest in functional languages such as Id [21] and Sisal [10]. Smart compilers will be used which do global analysis of the program. For the near future, the data-parallel model of computation, which is implemented both on SIMD and MIMD machines, will be very important.

In contrast, floating-point operations are not so crucial in a processor for a workstation or server. Simple pipelines are feasible and adequate. Only the local memory latency is really important. Even if you have a couple of processors, they're going to be on a bus, and not have the latency problems associated with big networks. The main mechanism for tolerating latency will be the cache, and the use of the other mechanisms becomes not so important.

In parallel scientific computation, parallel threads will arise from the same program, or even the same module in the program, threads that are simultaneously active on a processor. But in a workstation, it's most likely that, if you have a multithreaded processor, the threads will arise from different jobs or different processes, giving you a very different kind of problem to solve.

Most specification of parallelism, in existing workstation jobs, is explicit; it's done using operating system primitives. Compilation is by direct translation into machine code plus local optimizations. It is less important that the compiler be able to recognize parallelism on the same scale as in scientific computation.

So the requirements are very different, and therefore, I would expect that the kind of multithreaded processing element for the two domains could be extremely different. And there's no reason why we should expect the same architecture, or same design of multithreaded processor, to be really ideally appropriate for both domains.

So I think there's an opportunity for quite a variety of work, I think—a variety of solutions. Here are some short-term goals for multithreading:

- Design a multithreaded processor suitable for workstations and servers.
- Design a multithreaded processor which is well-suited to the needs of high-performance parallel scientific computation.

Long term, there is a more ambitious goal. We'd like to see the two domains merge, so you really *could* attack both problems, both domains of application, with the same kind of hardware. Why would we want it to merge? Because we really would like to use the same powerful languages in both domains. We would like to use the same implementations. We would like to be able to *move* programs from workstations to supercomputers and vice versa.

But there are a couple of big problems. One is that we need to have a sufficiently general program model that will cover both areas, both domains. A second big and nasty problem is having a virtual memory concept that will be viable on multiprocessors and will execute efficiently and support an adequate general program model. That's a big challenge.

HALSTEAD: Thank you, Jack. Now we'll turn the microphone over to Nikhil.

NIKHIL: I'm also very much of the opinion that multithreading is largely a compilation issue. Even if you *gave* me a multithreaded processor, it's very unlikely that you're really going to give me as many threads in hardware as I really need. I'm going to be multiplexing them anyway. So whether you give me 1 or N doesn't matter to me very much.

However, as a first approximation, I would say that it doesn't weaken my compilation problem. The resource management problem doesn't change in the first approximation. But what multithreaded architectures *can* do, because they balance communications better with communication, is to *simplify* a lot of my compilation and resource management problems.

So I'll start (and I say this independent of language models), and I'm going to repeat a point I've made before, which is really to think of languages and architectures separately.

[Referring to a diagram] I have my 2×2 crossbar switch here, and on this side you see an approach that's very popular today: either starting directly from a message-passing code and going to a large single-threaded multi [such as nCUBE, iPSC, or CM-5], or starting at a slightly higher level [e.g., Fortran 90] and compiling into code for that.

Now my point is that, even for *this* path, switching to a different architecture (that is, compiling conventional code to a multithreaded architecture) is a very viable and attractive alternative, and, even with no serious change in your compilation or resource management, the better communications here should help you. But actually, it may even simplify your life *substantially* here, because, in this compilation path, one of the major things that occupies your mind is: do few fat communications, because you want to avoid communications as much as you can, and that pressure is much lower when you go through this path up here. So in that sense, I see this as a perfectly natural thing to do and simplifying the compilation and resource management problems there.

In the rest of my talk, I'm really going to concentrate more on being able to support more *general* programming models. Already, we go to some intermediate forms, and today, compile to these kinds of single-threaded multis (or we're in the *process* of writing compilers like this). And we do this by multiplexing virtual threads in a single context. Going to multithreaded machines is just going to increase performance for me, and it may simplify a few things for me.

So I'll say a few more things there.

Now first of all, where is the parallelism going to come from? I think that it's basically going to come from this class of languages [Id [21], Sisal [10], parallel object-oriented languages, etc.]. (Now, I think David Callahan earlier said "explicit," but I found that the terms "explicit" and "implicit" parallelism are like lattices: some people view them this way and some people view them that way, and there are people who claim that Id is explicitly parallel, and so on. But I think you understand what I mean when I list the languages up here.) And I think at least for *massive* parallelism, having explicit thread management, like thread packages, is not really very viable.

Compilation issues: Now it seems to me, in general, for exploiting multithreaded architectures *well*, we need some good, fine-grain, parallel intermediate languages that are executable. In other words, they have an operational semantics so you can reason about them, transform them, prove equivalence between them, and so on. There are a number of existing research proposals along these lines.

However, there seem to be these two groups, A and B, where, today at least, A and B seem to me to be roughly equivalent in expressive power, but the analyses and transformations being done on them are quite different. From A, we start with extremely parallel graphs, and we try to produce less-parallel graphs with longer and fewer threads. And in B, starting typically from Fortran, we start with one long thread, and the purpose of analysis and transformation is to produce graphs with more parallelism, with shorter and more threads.

Now I'm very optimistic of us seeing the same kind of merging between the imperative and (previously) functional styles coming together, that we saw at the architectural level, happening even at the compiler level. And I think a sign of that is at least we're using graphs, now, that seem to have similar semantics out here, in the intermediate form.

IANNUCCI: I'll just mention that Kudos [9] defies that classification. Kudos doesn't really follow either classification B or classification A.

NIKHIL: Well, perhaps you could educate us on why that's the case.

OK, with regard to resource management, this is what I mean by saying, "to first order, multithreading does not affect me very much." The first-order problem that I'd like to understand and solve is: where do I place activation frames amongst the nodes of my machine? Where do I place heap objects? How do I recycle objects? How do I do garbage collection? And those problems are going to be equally important whether or not the processor is single- or multithreaded. What *might* happen is that, because of increased speeds due to multithreading, the pressure for good solutions is going to be larger, because I'm going to be *consuming* resources faster. But the same problem would occur even if you just replace my single-thread processor by one twice as fast.

Now, one thing I should point out about interleaved threads, is that we've seen multithreaded architectures today that at some high level you think of as multithreaded, but they have different approaches. Some of them *really* interleave threads, so that *really*, things are running in parallel. Whereas others, you run one thread exclusively, until it stops at some suspension or termination point, and switch to another thread. When you do interleaving, local resource management itself starts becoming a little challenging, because resource managers typically have critical sections, for which you will need exclusive access. And I'll leave it at that, without going into more detail.

Scheduling and locality: again, scheduling inter-*node* things are a first-order problem as far as I'm concerned. And that's not affected by multithreading. *Intra*-node is becoming more interesting, and here it is tied in a lot with what David Culler was talking about, which is that many multithreaded architectures do not give me control over selection of which thread to do next. And that can actually make resource management much more complicated in a multithreaded processor.

So multithreading can simplify things for me because of a better balance between computation and communication, and therefore, because of that better balance, I can not only support these two models which are very popular already, but go to more general and more easy-to-program models. And so, in that sense, multithreaded architectures may actually get me to universal, general-purpose computation models.

HALSTEAD: Thank you, Nikhil. I'll turn the microphone over to Vivek Sarkar.

SARKAR: People like this are a more reasonable bunch than hardware people. We've been quite peaceful so far. And I'll see if I can stir up some controversy during these five minutes.

One of the problems I've had during this day's discussions is that many questions were asked which, I think, were sort of orthogonal to the issue of parallelism, or at least multiprocessor parallelism. They are issues that would arise on a uniprocessor, too. So I think we've got to be a little careful. If we haven't solved a problem for a uniprocessor, we shouldn't go hounding people and asking them if they've solved it for a parallel machine.

So, for instance, do multithreaded architectures need extra parallelism? Well, it turns out that all architectures need more than P-way parallelism to exploit P processors, and this is also true when $P = 1$.

Let's take an example. This is the kind of thing that makes RISC processors run fast, and here's an example where we've transformed the matrix-multiply loop by unrolling the two outer i and j loops twice each, and putting the body in the innermost position. And this is purely to improve the sequential execution time on an RS/6000 processor. And the result is, you save on locality (the loads and stores of the array variable C, that accumulate the sum, that's been replaced by a scalar variable), and you get this nice tight loop, where multiply-add is a single instruction, Essentially, for each iteration of the original matrix multiply, you just do one load, one arithmetic operation.

But to be able to do this, you're giving up some extra parallelism. Those four statements could potentially, in an idle sense, have been executed on different processors. So I think this issue is general, and it certainly applies to multithreaded architectures as well. That is, you do need to use more parallelism than the number of processors.

This also shows why all the work in automatic parallelization of Fortran, and so on, can actually be important. Even if it doesn't speed up a program on multiple processors, it's the kind of thing you have to depend on to get peak performance in the single processor. To be able to do this replacement, you had to have been able to do the data dependence analysis that tells you you can replace the array variable by a scalar, and that tells you that you can unroll the loops.

So what about locality? Now again, in the uniprocessor world, at least this RISC high-performance processor world, cache is something we do depend on a lot, and if you use a cache in a multithreaded architecture, it seems to me you'll have less latency to hide, so you'll require fewer threads.

The way this would scale: in my opinion, I think cache consistency is too strong a requirement. I don't think you really need to have consistent caches when you go parallel. All you need is the memory to be consistent, if there are parallel activities in different processors, operating on their own caches. On the whole business of false sharing, the fact that it's false should tell you that there's no need to keep them consistent. You just need to be able somehow to merge them in main memory, so that you get consistent memory. But the processors should be able to access cache at full speed.

So this is another thing which you can pick up from the uniprocessor world and take over to multiprocessors and multithreaded architectures. And responding to a comment made by Burton earlier, I think we've grown to love line sizes that are greater than one, and I've actually used them quite well for reducing locality costs. You pay the memory access cost of eight to ten cycles, and get eight to ten words, and the compiler just works on restructuring your code so that you reuse those words in cache. People had mentioned earlier matrix transpose as an example. We find good locality in matrix transpose for this reason. And the best way to run matrix transpose is to block it. And that just falls out from the kinds of compiler optimizations we do.

Just a quick mention about where the parallelism will come from, and then I'll end with what I think the two important problems would be. The parallelism will come from the same kind of sources that you've heard earlier. Automatic parallelization will just give you some, but it won't cut it at all for large-scale parallelism. A real language is nice, because the user doesn't have to play around with tasks; it's much better than the X3H5 parallel Fortran kind of proposals. New algorithms: well, I think the trend is very much the more implicit, the cleaner, the smaller the language, the better.

All of this program parallelism can be represented in an architecture-independent way. We worked a lot with the program dependence graph that represents parallelism in Fortran. It *is* a matter of fact that we haven't found much parallelism in Fortran programs, but that representation was very similar to the representation of dataflow graphs, as Nikhil pointed out. And all the optimizations and transformations, thread partitioning, and so on, can work on that same uniform representation.

Whether you get all that parallelism or not depends on what source you use, how you write your original algorithm, and whether the parallelism is there. But I certainly think that these two things should merge together, if they haven't already.

And finally (since Bert is getting up)...

Of the two kinds of problems that I'm interested in, the second I think is more close at hand. It's sort of a thread-partitioning kind of problem, and it's something I worked on for Sisal in my thesis work [23]. And again, if you at least look at the cases where you know how to run the program efficiently on a uniprocessor, this would mean: let's avoid the issue of lazy and lenient evaluation, and so on, where just uniprocessor execution is hard. It is on *those* cases, I claim, that you can do an efficient job of thread-partitioning just by developing this framework, similar to what I've done in terms of estimating execution time costs, and using the dependences to pick out the threads. I don't believe this is the large-scale scheduling that David talked about. This is just forming the right kind of threads to balance the granularity and locality.

The first one is something that's really more hard for your compiler people. All our compiler optimizations are based on the notion of a sequential control flow graph. And especially if we go to multithreaded architectures, we've got to make this thread a first-class citizen in our whole framework. And a logical follow-on to the program dependence graph is to think of a parallel control flow graph, and when you branch on a label, you branch to multiple threads. And we need to extend all our technology (dominators, static single-assignment form, loop transformations, everything) to figure out how they would be safe in this kind of environment. So that's more of an open problem.

That's all I have to say.

HALSTEAD: Thank you very much, Vivek.

All right, now here's where we perform the alertness test on the audience. Did anybody from the audience have comments, challenges, provocative comments, or other barbs to launch at our panel?

CALLAHAN: [Pause] You fail!

HALSTEAD: Much as I suspected. Yes?

BÖHM: I have a question about language issues versus resource management. How does the non-strictness of Id [21] influence resource management, and what kind of research is needed to solve the problems here? I would say Nikhil...

NIKHIL: Well, I think it makes scheduling much more difficult. For example, if you have a loop, then non-strict evaluation allows you to run ahead allocating resources, unfolding this computation, or in a recursive function call or something like that, which sits there idle until the computation actually catches up with that point in the unfolding of the tree.

So I think that's a much more serious problem because of non-strictness. If you have a strict language, you can be much more aggressive about throttling the way you unfold a computation, so I think you have a much better handle on resource management.

So it's a tradeoff. On the one hand, it becomes much more hungry for resources because of non-strictness, but on the other hand, it also gets you an enormous amount of parallelism, in a few places where the other kinds of strict evaluation do not.

I don't know how to manage that demon yet. It's quite a challenging problem.

BÖHM: Do you see any way of carving that problem up, even in a non-strict language, or is it just too hard?

NIKHIL: In a non-strict language, I think it's very hard. I don't know whether it's too hard, but we may be going too far. But a good compromise would be to start with a strict language, but use non-strict evaluation for the parallelism. So that basically, the idea is the compiler can make safe assumptions about throttling, but whenever it wants to go ahead with the more aggressive unfolding, it can do so.

DENNIS: One thought, which is represented in the design of Sisal [10], is that an important role for non-strict evaluation is in supporting the producer-consumer paradigm of program execution. Sisal provides for this in stream data types, where any data type can be the element type for a stream. And you can even have streams of streams. Through the stream mechanism, you can implement producer-consumer computations. One possible position is that that takes care of all the needs one might have for non-strict evaluation. I think it's important to look at other places where non-strict evaluation is promoted and evaluate what the alternatives are.

BÖHM: Well, I can tell you that composing streams in Sisal is very costly. I really believe that when you're saying that you can get parallelism from non-strictness —

DENNIS: I don't think strictness of a language should be viewed as constraining the implementation. The implementation can use whatever evaluation techniques yield a consistent result. The fact that the language is so-called strict does not mean that the arguments cannot be evaluated or have to be evaluated before beginning evaluation of the function body.

ARVIND: But there *are* cases where a strict language *will* consume much more slowly. If we have a recursive data structure like a —

DENNIS: I think these cases *can* be tackled by smart compilation techniques. So it is not clear that you automatically get more storage requirement simply because the language is strict.

ARVIND: I think you *get* more storage requirement. You're saying, in *most* cases, compilers will be able to catch it, and that remains to be shown. Because if that can be shown, then resource manager problems for strict languages can also be finally solved.

DENNIS: Yes, I think that has actually been demonstrated for practical applications in the case of Sisal compilers.

HALSTEAD: I'd like to ask the panel: is there any difference between compiling for a multiprocessor made out of multithreaded processors and a multiprocessor made out of non-multithreaded processors?

NIKHIL: Not for me; my first diagram was exactly that. I go to exactly the same intermediate form; I do exactly the same optimizations. There are some back-end code-generation issues I think we might change, but whatever physical multiple contexts you give me, I will just treat them as N processors. In any case I'm doing my own multiplexing of virtual threads onto it. It doesn't make a big difference to me whether I'm going through a single-threaded multiprocessor versus a multithreaded multiprocessor.

DENNIS: The issue depends very crucially on the length of the threads that the machine can handle efficiently. The shorter the threads the machine can handle efficiently, the easier the compilation problem is.

SARKAR: Again, it depends on how different the multithreaded processor is from a single-threaded other processor. For instance, if the single-threaded processor has a cache, and the multithreaded processor does not have a cache, the compiler will suddenly have to do more work to overcome that locality, which it used to have in the non-multithreaded case. There are various other architectural features which, depending on whether they are included in the multithreaded case or not, the compiler may have to do more or less work.

NIKHIL: But those are (to me) orthogonal issues, about whether it has a cache or not. That is a separate decision from whether it is single-threaded or multithreaded. So, in that sense, that variation in compilation strategy is there between a single-threaded cacheless versus single-threaded cache-full multiprocessor.

SARKAR: Well, in practical terms, one can conceive of a multithreaded architecture without a cache (there's one being built!), and that is because the multiple threads can be used to hide the latency. But to make that happen, you have to do extra compiler work.

HALSTEAD: I think the person on the panel who's probably spent the *most* time thinking about this hasn't said anything yet. David?

CALLAHAN: Well, one of the distinctions when you say "P processors" versus "multiplexing the same number of logical streams P on a smaller number of physical resources" gives rise to the following question: That in the P case, in regions of lower parallelism, are those processors sitting idle? And if they are, you're now willing to expend speculative work to try to give them something to do in the hope of speeding up the main computation. You'll do redundant work in order to achieve parallelism, as is typical in parallel prefetching.

If, however, you're multiplexing the same number, these are all in the same physical resources, so the fact that you've reduced your logical number of threads doesn't mean you've reduced your processor utilization. Now you may be reluctant to add additional work, since raising your total work may raise your total time to complete the job.

So there's one real difference there. I mentioned this before as an area to work on.

AUDIENCE MEMBER: One question that I don't know if anybody's really posed before (probably have): if you take a particular algorithm or task to be done, there's probably a perfect amount of work you need to do to get that job done. The actual number of computations that you have to do. In some of the constructs I've seen for adding multithreading or dataflow, the amount of extra instructions executed to get a task done was probably more than the instructions that you would use as a general case. What type of measure would you expect we should have for how much extra work we should expect to do in getting this parallelism? Just like your talk, David, if we would be willing to do some speculative work to get the job done earlier —

CALLAHAN: Well, in general, of course, for a particular problem there's a range of space-time tradeoffs you can make in algorithm choice. Are you including *that*? I mean, typically that's not a compiler choice.

DENNIS: I think he wants to keep the algorithm fixed and then look at what happens to the parallelization.

Let me put it this way. If you look at the architectures, it *does* take some extra memory cycles to manage the threads, which would not be used in the case of a single-thread realization.

The argument is that there's some benefit to using those extra cycles. You may be making better use of memory, because there are objects in memory which are serving multiple functions during parallel evaluation. So their residence in memory is earning more in that case. This is an issue which is very often overlooked. Through parallel processing and multithreading, you are hopefully making better use of processing element resources.

CALLAHAN: Or the memory resources.

DENNIS: Well, both. So it's a balance. And whether it's a net payoff in terms of operations per dollar invested depends on details.

CALLAHAN: Hard to predict.

HALSTEAD: I think there was a question back there.

AUDIENCE MEMBER: Well, going back to what you were saying about would you compile differently for a multithreaded machine, it seems clear that if the compiler needs to pass the information along to a machine (like Alewife [1]) with a cache, you might want to think about which threads you're going to map to the same processor to try to avoid interference in the cache.

NIKHIL: The main difference I see in compiling for multithreaded versus single-threaded is the degree of effort I will put into trying to group together messages. And that's whether I'm working with Id [21] or Fortran-90; the issue there is really the same. The balance is different, and so the optimizations are going to be different.

DENNIS: If you're thinking of high-performance scientific computation, if you take SAXPY, and divide it into threads, and run it on a multithreaded machine, the footprint from the cache is going to be the same as if you ran that on a single thread, right? So the locality issue is going to be the same.

HALSTEAD: Anant, did you still have a question?

AGARWAL: Yeah, I guess from a long time ago I had a comment.

I think it's very neat allowing multithreading, allowing some extra hardware support in the processor. I think that's very neat, if the compiler gets another axis along which optimization can be done. But I think it's a more challenging problem. Any problems with that?

CALLAHAN: I think I agree with that, but it sort of depends on what you're comparing it to. If you're comparing compiling for multithread versus compiling for a SPARC or conventional RISC, then sure, it's harder, because now we've got another way of expressing parallelism. But if you're comparing multithread with the same number of physical processors with some mechanism to control them, it's not clear that it's that different.

NIKHIL: And, having that extra axis, we make things more orthogonal, and therefore simpler. So having more axes doesn't necessarily mean more complex.

CALLAHAN: This one does.

AGARWAL: I really don't understand when you say that. The parameters of these two axes are different. One is gains in performance, and the other is gains in cost-effectiveness. It *is* a harder problem.

NIKHIL: That axis, right. But it may simplify some things substantially on another axis, such as how you optimize messages.

HALSTEAD: Gao?

GAO: Consider the Class B applications, which were mentioned eariler.[3] Recently, I had a long discussion with someone from the West Coast. The following question was raised: in a Class B application, is there enough parallelism to justify multithreaded at all? Because one study, quoted again and again, from the West Coast, is the David Wall and Jouppi paper on instruction-level parallelism [14]. Even if you have the *best* compiler, which has perfect alias analysis and everything, have register renaming or whatever architectural support, 100% branch prediction, and so on and so forth, the parallelism is going to be two, three, up to seven, although they have doubled it in the last two years. So therefore, I'd like to —

CALLAHAN: So you say we just should wait!

[3]The workstation domain described by Jack Dennis in his opening statement on page 27.

GAO: My question to the panel is that, first of all, is this study relevant to the discussion? Secondly, do you think either the language, compiler, or all of them working together, actually can make *more* than that, for this class of applications (workstation-type applications)? Otherwise, there is just no point in building a multithreaded architecture.

DENNIS: Oh, I disagree. The point is that, the threads on a workstation will arise from the fact that the workstation is going to be doing different processes and jobs simultaneously. And the compiler has nothing to do with that, because the creation and operation of a thread is done through the operating system. Now the compiler won't be involved until we get to the point that a new *programming* model is appropriate for these workstations. I don't know how that's going to come about, because this is an area which is really going to be very conservative as we move into the future.

GAO: So essentially, your point is that parallelism has to come from different jobs on a workstation.

DENNIS: Well, at the present time, the parallelism in workstations *is* coming from different processes. The processes may be working for the same job; they may be Unix$^{\mathrm{TM}}$ forks, you see, or pipes, or whatever. And there'll be a lot of them. But it arises from the operating system, not from parallelism that's detected by the compiler.

NIKHIL: I would like to repeat something which (I think) Bill mentioned this morning. I view these studies, in some sense, as self-fulfilling prophesies. That is, until you give me good multithreaded architectures, I'm not going to write programs in a style that is multithreaded, or compile in that manner, and then I take those traces and I run simulations and say "see, there's no parallelism." So I really don't believe them very much.

HALSTEAD: Charles?

LEISERSON: I'm hearing a lot on the distinction between multithreaded and single-threaded, but isn't one of the big competitions in parallel computing also with vector processors? Even the CM-5 has vector units on it. And I'm wondering whether, or do you folks think about whether, you can have a multithreaded vector machine, or whether multithreading obviates the *need* for vectors. What happens if you take these two different architectural ideas? What are you going to do?

CALLAHAN: First, allow me to make a couple of comments about vectors. One is that, one of the early motivations for vector instructions was sort of an instruction-fetch–rate problem, in addition to other things, as well as memory latency. And that instruction fetch problem is still going to be there. You know one of the reasons we have wide instructions is because it's hard to get enough there, and so every time you get one, you want to make good use of it.

DENNIS: Isn't it solved by an instruction *cache*?

CALLAHAN: No. Not well enough.

DENNIS: Hmmmm...

CALLAHAN: Hmmmm! Well, what do you say about the Cray? On the Cray (I think it's the Y), the scalar issue rate is about every other clock, and mostly due to instruction-fetch problems.

The other reason for vectors is to get a lot of loads in the air so you can mask those latencies. And I think *that* problem is basically gone; we can get a lot of loads in the air. We may still have a problem with instruction fetch.

ARVIND: I guess you're including instruction dispatch also with instruction fetch.

CALLAHAN: Oh yeah, the whole cycle. If I've just done one, now I've got to go find another one —

ARVIND: That I can imagine. Otherwise, if it's simply a question of fetching it —

CALLAHAN: Well, there's fetching, and there's the potential for branch, there's condition codes, and there's all this stuff that goes on too.

DENNIS: Well, I stand on what I said in 1988, which was that vector processing is obsolete.

CALLAHAN: And I *will* stand by it.

DENNIS: OK, fine, I have some company. For instance, in the CM-5 there's this strange artifact of its SIMD history.

NIKHIL: Well, I think there's another motivation for having vector processors, which is, if you have P processors and vector length L, and you have to claim a teraflop, you have to have $\frac{1}{PL}$ vector units in your machine.

DENNIS: That's interesting.

HALSTEAD: I think there are a lot of issues that are related to this whole area of programming, compilation and resource management that we haven't really been able to touch on in this discussion. And I for one would like to see these being a subject of a lot more work in the future, that could be reported on in a future workshop.

Unfortunately, we have our own little resource management issue, dealing with the clock right now. And so what I would like to do is to finish with another of Bob Iannucci's inimitable questions, that we can perhaps go down the panel and get a final word from each member. And that is: discuss the techno-economic interactions among the following architectural features (and we already got started on this): (1) multithreading, (2) vector instructions, and two other hot topics in computer architecture, namely, (3) out-of-order instruction issue and (4) Very Long Instruction Words. Are these all forms of the same thing? Are there any differences among them as far as the software techniques for making use of them are concerned?

So maybe we can get a nugget of wisdom (possibly one bit) out of each panelist, on this subject, and we can declare the workshop a success and go home. So let's just start from this end. Nikhil?

NIKHIL: I don't understand the question.

DENNIS: Well, I'll give you a two-bit answer, all right? I'll select (1) over the other three.

NIKHIL: I'd say that (1), (3), and (4) are sort of compatible things, in the sense that I can think of them as different levels of abstraction when compiling. But I don't know how to fit (2) into that.

SARKAR: OK, well, I'm going to be a subset in between Nikhil and Jack. I think one is that I'd select (1) and (3). I think vectors are obsolete, especially with, again, as I said, my preference for cache line sizes greater than one. That's as long as the Cray single vector registers will give you that facility. And (4), again, I think instruction caches solve the instruction-fetch problem for large instruction words. So I think (1) and (3) are really the two that will interact in the future.

CALLAHAN: Just one more comment I might make about the wide instructions. It may be sort of a poor man's way of getting this thread scheduling that Dave Culler has been working on, in a sense. I've gotten several things together, now I want to issue several things before I wait again, and they work well in that context.

HALSTEAD: Well, with that, I'd like to thank the panel, and thank the audience. I wish you all good trips home and look forward to seeing you all again.

4 Acknowledgments

The panelists and workshop organizers have given permission for the use of these transcripts, and were very helpful in checking their accuracy. Govind Ramaswamy and Bhama Sridharan of ACAPS also helped to interpret some parts of the tape.

References

[1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 18(2), June 1990.

[2] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International*

Conference on Supercomputing, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM. Also in *Computer Architecture News*, 18(3), September 1990.

[4] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 213–223, Williamsburg, Virginia, April 21–24, 1991. ACM SIGPLAN. Also in *SIGPLAN Notices*, 26(7), July 1991.

[5] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[6] David E. Culler and Gregory M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, December 1990.

[7] David E. Culler, Anurag Sah, Klaus Eric Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[8] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[9] Kattamuri Ekanadham. Kudos. In *Proceedings of the 1st Annual Ohio State University Workshop on Parallel Computing*, Columbus, Ohio, March 21–23, 1990.

[10] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[11] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report 90-141 (Rice) and SCCS-42c (Syracuse), Department of Computer Science, Rice University and Center for Computational Science, Syracuse University, April 1991.

[12] R. H. Halstead Jr and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. Also in *Computer Architecture News*, 16(2), May 1988.

[13] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[14] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, pages 272–282, Boston, Massachusetts, April 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. Also in *Computer Architecture News*, 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices*, 24, May 1989.

[15] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 105–112, Cambridge, Massachusetts, August 4–6, 1986. ACM SIGPLAN, SIGACT, and SIGART.

[16] Janusz S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and its Applications*. Scientific Computation Series. MIT Press, Cambridge, Massachusetts, 1985.

[17] Monica S. Lam and Martin C. Rinard. Course-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 94–105, Williamsburg, Virginia, April 21–24, 1991. ACM SIGPLAN. Also in *SIGPLAN Notices*, 26(7), July 1991.

[18] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 20(2), May 1992.

[19] D. May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[20] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 20(2), May 1992.

[21] Rishiyur Sivaswami Nikhil. The parallel programming language Id and its compilation for parallel machines. *International Journal of High Speed Computing*, 5(2):171–223, 1993.

[22] P. R. Nuth and W. J. Dally. A mechanism for efficient context switching. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 301–304, Cambridge, Massachusetts, October 14–16, 1991.

[23] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. Revised version of the author's Ph.D. dissertation (Stanford University, April 1987).

[24] Thorsten von Eicken, David E. Culler, Sech Copen Goldstein, and Klaus Eric Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. Also in *Computer Architecture News*, 20(2), May 1992.

The Workshop on Multithreaded Computers – Agenda

Friday, November 22, 1991
Supercomputing '91
Albuquerque Convention Center

Morning Session (8:15 – 12:15):

8:15: Welcome

Paper Session I (8:30 – 11:15):

Session chair: Guang R. Gao
All morning talks are 40 minutes with 5 minutes Q&A.

8:30: Arvind (MIT)
*From Monsoon to *T*

9:15: Anant Agarwal (MIT)
The Architecture and Design of Sparcle

10:00: Coffee break

10:30: Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, Burton Smith (Tera Computer)
Exploiting Heterogeneous Parallelism on a Multi-Thread Multiprocessor

11:15: **Panel Session I (11:15 – 12:15):**

Panel Chair: Bob Iannucci
Architectural and Implementation Issues for Multithreading

Panelists:

Anant Agarwal, MIT
Bill Dally, MIT
Anoop Gupta, Stanford
Greg Papdopoulos, MIT
Burton Smith, Tera Computer

12:15: Lunch (1 hour, 15 minutes)

The Workshop on Multithreaded Computers – Agenda (continued)

Afternoon Session (1:30 – 5:00):

Paper Session II (1:30 – 4:00):

Session chair: Bert Halstead
All afternoon talks are 25 minutes with 5 minutes Q&A.

1:30: Shuichi Sakai, Yuetsu Kodama, Mitsuhisa Sato, Yoshinori Yamaguchi (Electrotechnical Laboratory)
Current Status of EM-4 and EM-5

2:00: Peter Nuth (MIT)
Named State and Efficient Context Switching

2:30: Herbert Hum (McGill University)
The Super-Actor Machine: A Novel Hybrid Dataflow/von Neumann Architecture

3:00: Coffee Break

3:30: David Culler (UC Berkeley)
Multithreading as a Compilation Discipline

4:00: **Panel Session II (4:00 – 5:00):**

Panel Chair: Bert Halstead
Programming, Compilation, and Resource Management Issues for Multithreading

Panelists:

David Callahan, Tera Computer
Jack Dennis, MIT
Rishiyur S. Nikhil, DEC
Vivek Sarkar, IBM

5:00: Close