Communication Thread Cache: High-speed Communication Support for SMT

Kang, Chang Woo

Abstract—This report is a part of the fulfillment for EE653 class project. A novel approach to support high-speed communication for SMT architecture is proposed, so called communication thread cache. The cache will store the status of threads, which have been blocked due to the latency of the local or remote memory access and reactivate the sleeping threads as soon as data is available. This mechanism provides not only a fast notification scheme for network interface but also a simple way for threads to sleep for the fine-grain synchronization[1]. Highly communication oriented applications and active network systems will be able to become good examples using communication thread cache.

I. Introduction

In current high-performance multi-computer systems, we have faced two major problems: memory wall problem and overhead in network interface. First, memory wall problem is still under the main attack by numerous computer architects. Second, the fast communication support became the essential part to achieve high-performance computing in multi-computer system as the network speed in the physical layer is fast enough to choke network interface system. Semiconductor technology has been growing so fast such that millions of transistors can be fabricated on a chip now. Then, a question dawns on us. “What can we do in order to solve these problems if we can have enough resources on a chip even with DRAM modules?” Answers are quite straightforward with embedded DRAM technology. People are trying to put processing logic with DRAM modules in a chip so that the latency to access the memory becomes short [2], but the demand for the size of memory increases as the environment of applications becomes dynamic and complicated, resulting in another external memory through off-chip communication. Unfortunately, we won’t be able to figure out this problem clearly in a decade without a new renovation on the memory device technology. On the other hand, computer architects developed a new architecture, called SMT(simultaneous multithreading), in order to provide tolerance for the memory latency by feeding instructions from different threads while a thread, which accesses memory, does not issue instructions. The SMT architecture showed up as the next generation processor, which has advantages of cost effective design and fully exploitation of TLP and ILP. Multiple program counters and register files will be added into the currently fully-fledged superscalar processor so that multiple contexts can reside on a CPU. By allowing multiple threads to be active simultaneously, inherently TLP (thread level parallelism can be achieved, and ILP (instruction level parallelism) will be fully exploited, seeing that SMT is designed based on superscalar processor. It is very clear that multiple threads will run and sleep on a chip dynamically in the near future computer system. In the era of millions of transistors, we can simply think of the

Parallel & Distributed Processing Center/USC

embedded network interface with a CPU on a chip to work out the overhead on the network interface. However, we need to optimize it by means of fully making use of advantages of SMT architecture. Multi threads are active in SMT, but conversely this means that multi threads can sleep when they are blocked by some specific operations causing delay for the next operation such as local memory access or communication between processors. This report will investigate a method to improve the notification time in network interface mainly by means of making use of inherent SMT advantages and sufficient resource such as transistors.

The rest of this report is organized as follows: Section II gives the background of network interface. Communication thread cache will be introduced in section III. In section IV, a new architecture with the idea will be proposed. Two examples for supporting the idea will be provided in section V. The basic idea of modeling the network system with the communication thread cache will be described in section VI. In section VII, future work will be discussed for the simulation. I finally conclude in section VIII.

II. PROBLEM STATEMENT AND BACKGROUND

A. Problem Statement

The network interface became a bottleneck in multi-computer systems, which heavily rely on message passing mechanisms. Particularly, the notification time could not be reduced much, even though other overhead could be improved. The speed of physical network itself becomes pretty fast due to the new links such as optical fibers, causing heavy traffic through the narrow, long path to the node computer, which is the network interface. Once a packet arrives at a target node, it must wait until it is processed in order of arrival. There are several overheads relevant to the message handling. Those are copying operations between kernel and user area, context switching, notification of message arrival, and start-up cost due to the network buffer management. In the near future, the high-performance demand for the network system will be tremendous because not only the processor will try to send messages at GHz speed but also network links armed with optical fibers will move data at light speed. In addition, multi-nodes will be tightly coupled in the future system, and then the latency along the physical network will be ignorable. With a typical example in Figure 1, we can see where the overhead resides in a conventional computer system. Let's walk through the detail protocols for sending and receiving a message from P1 to P2.

1. User executes a send and traps the kernel on P1.
2. Kernel on P1 parses the message, makes a copy of the message in kernel space, and programs the output DMA.
3. Message is received by P2 and preprogrammed input DMA device dumps the message in some area in kernel space.
4. Input DMA device on P2 interrupts user and kernel is run on P2.

5. Kernel looks up the "send-rdy req" message and check to see if the receiver is blocked waiting for the message.
6. If so, the kernel on P2 programs an input DMA to receive the message and the output DMA to send a "recv-rdy reply" to P1; another user runs on P2.
7. P1 receives the "recv-rdy reply" through an input DMA. Input DMA interrupts user on P1 and kernel runs on P1.
8. Kernel on P1 programs output DMA to send message to P2; P1's DMA sends the message to P2.
9. Input DMA on P2 transfers the message to kernel memory and then interrupts the kernel.
10. The kernel on P2 moves the message to user space and then signals the user that the message received.

In case the receiver has not posted a receive yet in step5, the kernel reschedule a user. At the time the receiver executes the receive the kernel is trapped and the kernel will find that a send is pending. Then the kernel on P2 proceeds with step 6.

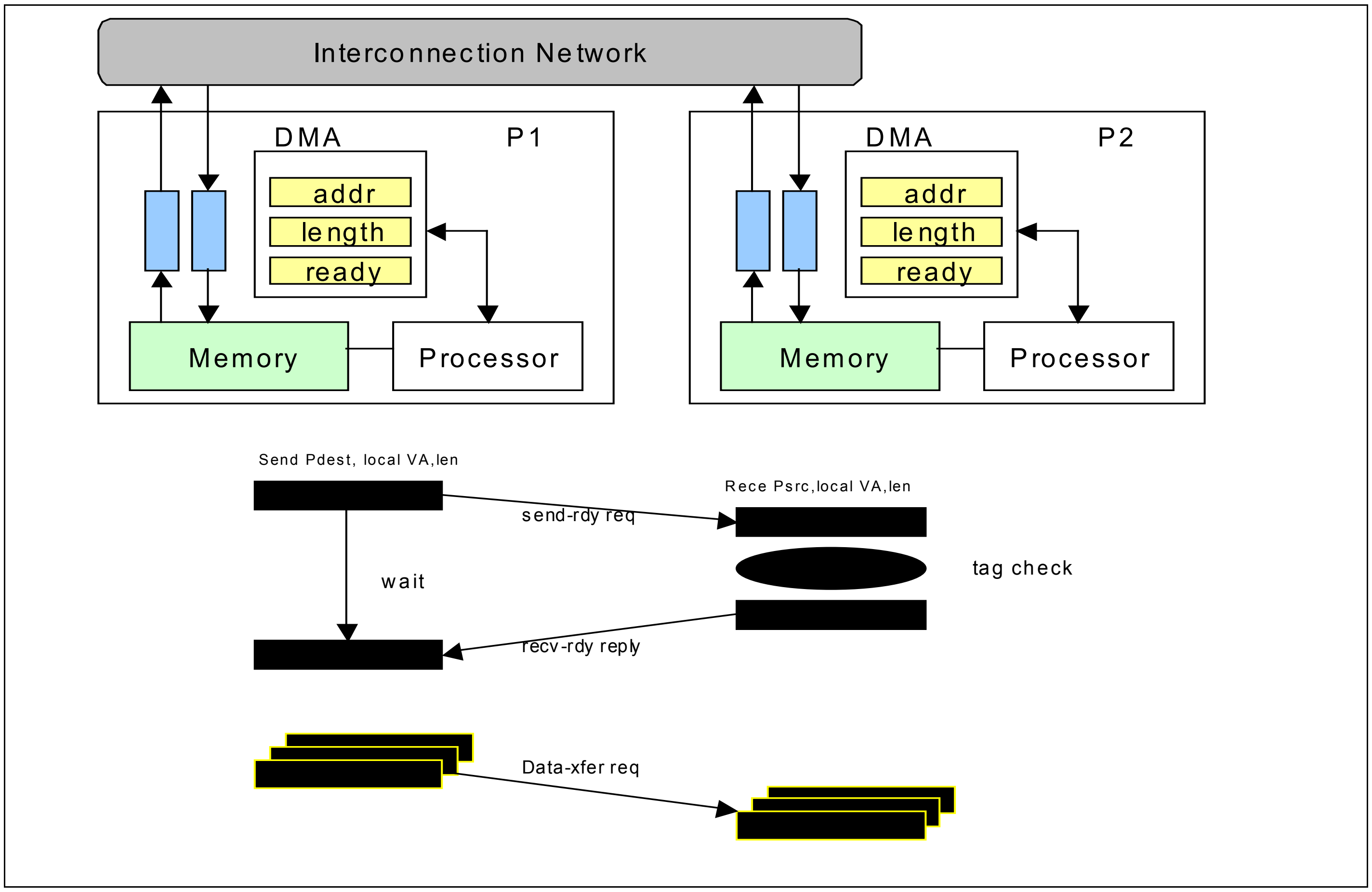

Figure 1 Synchronous physical DMA message passing

In this very primitive example, four memory copies between user space and kernel space occurred. Eight context switches in both side involves four storing contexts and four restoring contexts. When a message arrives at the P2, an interrupt must be introduced, causing long notification time. Because messages must be copied into the network buffer and the buffer must be managed properly before sending them, the start-up cost cannot be ignorable. In fact, the start-up cost was the biggest stumbling block on the network path. In this research, I concentrate on the reduction of notification time in tightly coupled SMT architecture.

B. User-level Network Interface

Kernel intervention causes a considerable amount of data movement between kernel space and user space. Therefore, researches to get rid of copying into kernel space were main issues in this field. User-level network interface allows user processes to access NI (network interface) hardware resources such as control registers and network buffers directly [3]. By using this scheme, the number of memory copies could be reduced depending on the place in which the network interface is connected. According to the research conducted by [3], messages do not have to be stored into the network buffer in the memory system directly as the network interface has big memory in it and is connected to the memory bus in Figure 2. This figure shows the evolution of network interface system with respect to the position of connection and the message movement.

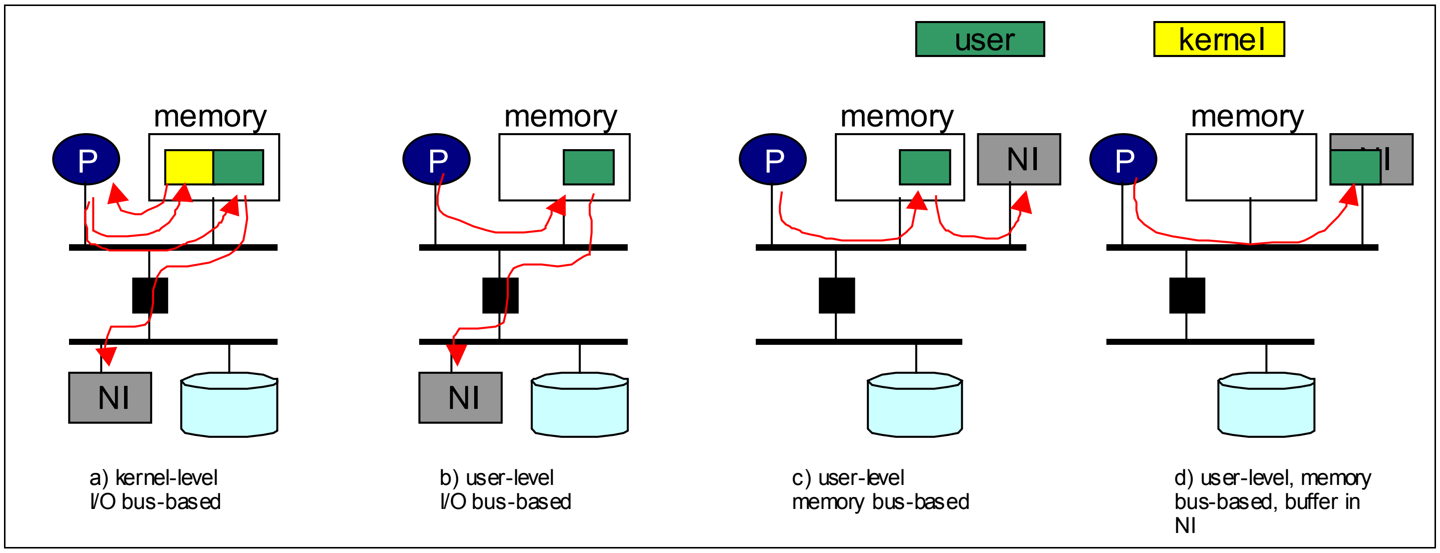
user
kernel
memory
P
NI
a) kernel-level I/O bus-based
b) user-level I/O bus-based
c) user-level memory bus-based
d) user-level, memory bus-based, buffer in NI

Figure 2. Evolution of network interface access

In the case of kernel-level access and I/O bus-based connection, four times of memory access is indispensable including load and store operation. However, as the network interface is accessible by user processes and is connected to the memory bus so as to get high bandwidth, the number of copying operation has been reduced to two times in case c and d in Figure 2. Furthermore, network interface might have enough buffer space to store incoming and outgoing messages as in case d, and then messages can go to the network interface directly without stalling due to memory access.

C. Reducing the number of context switching

By allowing user processes to access network interface resources directly, context switch is not necessary anymore. In other words, zero context switch activity occurs in the sender side. In one of the following sections, we will see the overhead of each context switching.

D. Reducing the notification time

When a message arrives, it must notice its arrival to the processor by interrupting current work or being polled by the receiver process [4]. It is important to minimize the notification overhead of message arrival in order to reduce communication latency. Even though messages move so fast along the network wire, without good notification scheme the latency caused by late notification will hamper the overall performance. You almost got there so fast, but you must have waited in a long line.

Two methods are currently available: interrupt and polling. Interrupt is a primitive but expensive function which a processor provides. When a message arrives, it causes interrupt so that the processor recognizes the arrival and changes the context to the operating system. The operating system reads the message from the network buffer and saves it into the kernel space first. It checks which process has been waiting for that message and where it should go, and then it copies the message into the user space. A context switch occurs to the receiver process by invoking it. This is a quite heavy work, as every message should be handled in this manner. As shown in Table 1, the number of cycles involved with context switching between the kernel and a user process is about a thousand cycles [5]. Several techniques can be used to reduce this overhead by coding interrupt service routines in assembly language to use a small number of registers, switching to an alternate register set for interrupt processing, saving processor registers to unused floating-point registers, and providing on-chip memory for saving and restoring state[5]. On the other hand, this has an advantage of easy programming. Programmer does not have to worry about the latency of memory arrival to the network interface from the source since the message will notify it automatically.

Table 1. Interrupt const for responding and exiting

System Configuration	Response time	Exit time	Cycle(total)
220 MHz Strong ARM	2.1 us	1.2 us	726
200 MHz MIPS, 100 MHz SDRAM , and Full context saved	4.7 us	3.4 us	1540

The other way of notification is polling. The receiver process keeps checking a flag, which must be set as soon as message arrives, to identify its arrival. However, this checking process will consume processor clocks, meaning that the processor must keep staying on a small loop, which is a waste of hardware resource. This method will work fairly if the message arrival time is predictable and the size of packets is small. However, the arrival time is strongly dependent on the congestion in the network and relying on polling is quite dangerous if the message is lost in the middle of delivery. There are two scenarios of bad prediction of message arrival. If the message frequency is faster than polling frequency, the network will be congested and some of messages might be lost. On the other hand, if the message frequency is slower than polling frequency, CPU cycles must be wasted considerably. Both interrupt and polling schemes have advantages such as easy programming and cheap notification. In order to make use of advantages from them, a hybrid method has been proposed [6]. When a message arrives, a timer starts counting. If the message is not removed from the network through polling within a given amount of time, the watchdog interrupts the CPU. It sounds feasible but, in reality, setting the right time for the watchdog is problematic. Not only applications have different demands on message length, but also they have different behaviors in terms of networking. In addition, the watchdog timer is a limited resource and there will be an issue of how to share it between multiple processes. In a nutshell, a bad choice of the watchdog timeout value will eliminate most of the advantages.

E. Active messages

An active message has control information at the head of a message, which is the address of a user-level instruction sequence that will extract the message form the network and integrate it into the on-going computation [7]. The role of the user-level message handler is to get the message out of the network and into the computation ongoing on the processing node. This scheme reduces the start-up cost by eliminating the buffer management for the incoming and outgoing messages. Either storage for arriving data is pre-allocated in the use program or the message holds a simple request to which the handler can immediately reply, resulting in low start-up cost due to no buffer management. On the other hand, notification problem has not been solved as the arrived message has to interrupt the CPU and inherently it is hard to overlap computation and communication in real applications.

III. COMMUNICATON THREAD CACHE

In the era of millions of transistors, multiple threads on a chip must be a certain way to achieve high throughput and those threads must dynamically run on a chip in order to get fast reaction to the requests such as message arrival and external event from the outside world. However, the complexity of CPU increases, as it needs high performance [8, 9], causing the cost of context switching more expensive for precise interrupts. CPU must be always alert against every possible event. Modern environment for computer architecture is quite oriented to the computer networking and reducing the latency along the network path has became an important issue. Events caused by networking already started choking computer systems due to the high speed of message delivery and the slow process of those messages. For instance, even though a message arrived early and CPU is idling, the message cannot be processed fast enough not to block continuously incoming messages. It has go through all the steps for the precise interrupt such as storing state, handling events, and restoring state. On the other hand, SMT has an advantage of flexibility in choosing active threads, which issue instructions, among multiple threads, which are candidate threads for the next instruction issues. As a matter of fact, all of them do not have to be active as long as the other threads are fully making use of the shared resources such as functional units, instruction queue, branch prediction unit, and memory access units. In current computer system, the main cause of thread blocking is memory access regardless to local access or remote access through network system. Those threads can be in an inactive state and yield shared resources to the currently active threads. Without any expensive context switching operation, the sleeping threads can be reactivated as soon as their requests are satisfied. In the following section, I will describe this scheme in detail.

A. Definition

The CTC (*Communication Thread Cache*) is a cache, which stores information of blocked threads. The information will be the validity of a thread, the thread ID, the type of memory access, and the cause of blocking. As soon as a thread issues an instruction, which causes the long latency due to the remote memory access, it stores the information of the thread into the CTC and preserves contexts in the program counter and the register file as it is but invalidates the thread so as not to let the scheduler choose it as the next candidate for issuing instructions. When the required data arrives, the network interface notifies its arrival to the CTC. Identifying a right thread based upon thread ID and its validity is conducted in CTC and then the thread is activated again by validating as one of candidate for the next scheduling. The causes of blocking will be the local memory access as well as waiting operation for the message from remote nodes.

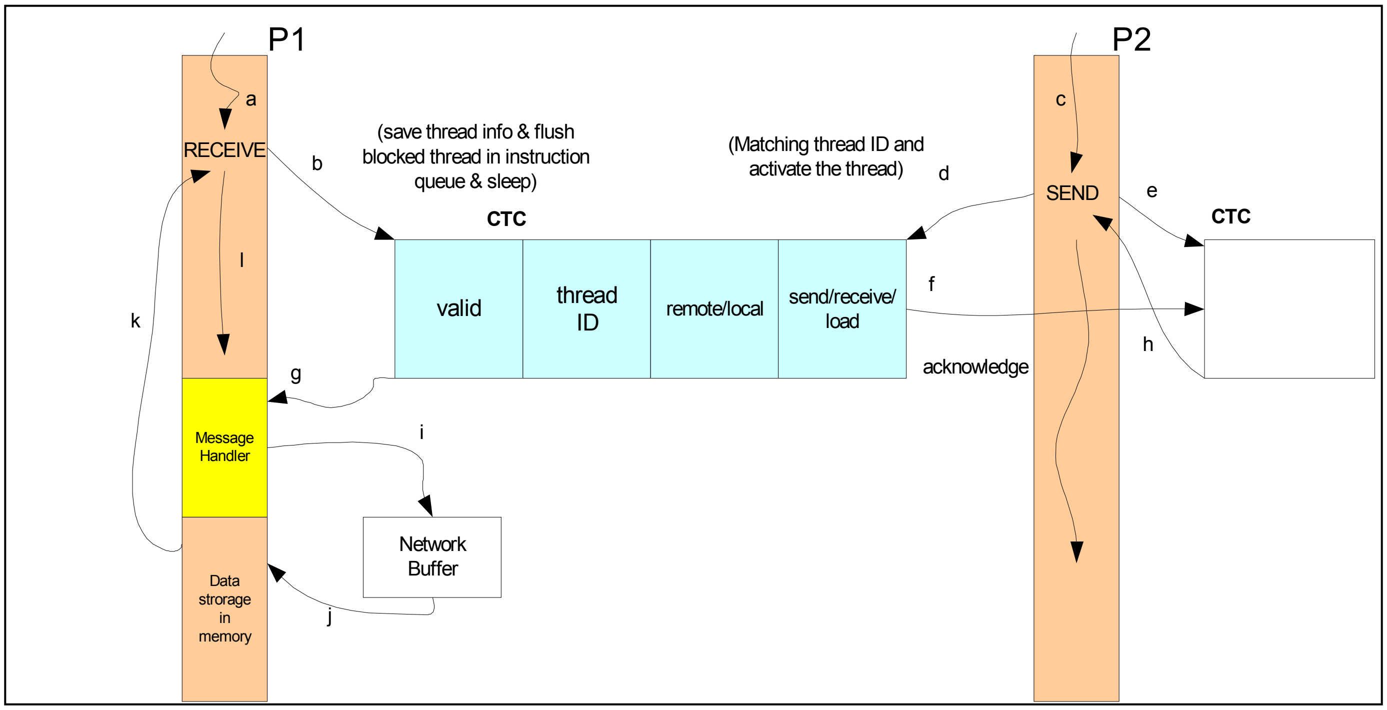

Figure 3. An example of message passing with CTC

In Figure 3, a procedure of message passing by using CTC is described. Process P1 issues RECEIVE to get data from the remote node. It stores the thread information into the CTC, flushes instruction queue in order to avoid the deadlock due to the blocked instruction, and invalidate its status so as not to be chosen as a thread for issuing next instructions. Process P2 issues SEND to deliver data to the remote process P1 and sleeps on CTC after the same procedure done for P1. As soon as the message arrives, the acknowledgement is returned to P2 so that it can recognize the safe arrival of the message and CTC activate P1 after verifying the identification of the sender. Of course, for the verification, both sender's and receiver's thread ID must be provided in CTC. The activate thread jumps to the message handler to fetch the message from the network buffer and returns to the next instruction of the RECEIVE to complete the program. By doing this, no interrupt is involved and blocked thread can be activated as soon as their demands are satisfied in short time.

B. *Thread scheduling*

If a thread is blocked by a long latency operation, it is invalidated and drop out of the candidates for utilizing SMT resources. All

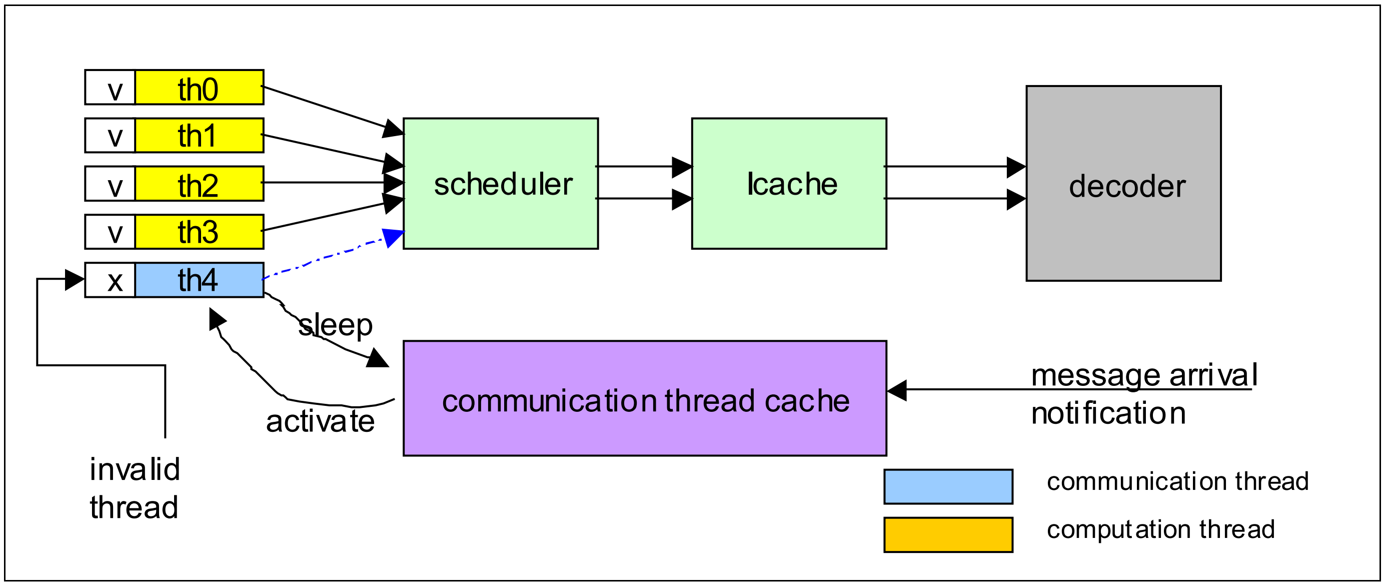

Figure 4. Thread scheduling with valid threads

of the instructions belonging to the thread will be flushed out of every functional unit and shared units to avoid blocking other threads proceeding. In Figure 4, five threads were active and thread 4 has been invalidated. Among four threads two threads will be chosen for issuing next instructions according to the strategy of scheduling such as priority order. By giving up functional resources and shared resources, other threads can fully make use of them. Of course, operating system must switch context of blocked threads when it keeps sleeping for a long time so that all of threads may not be invalid at the same moment. I introduce a terminology called *communication thread*, which is heavily supposed to communicate with other threads on other nodes. If one application consists of multiple threads, we can assign communication work on communication threads while computation threads compute data delivered by communication threads. This will be possible if fine-grain synchronization is supported inside the SMT [1].

C. Early arrival of a message

There will be a case that a message arrives early even before the receiver executes RECEIVE instruction. However, handling this case is very simple. We can set the valid bit and store sender's information in CTC. The receiver thread will invoke a user-level message handler as soon as it sees the valid bit and verifies its identity by comparing thread ID.

D. Protection

It is very important to guarantee the protection in network system. In user-level memory system, virtual memory can be used to protect individual address space. However, in this architecture, there might be a question whether a malicious thread can activate wrong threads intentionally or not. For this case, network interface can filter out intruders with a table, which has information of senders and receivers. Therefore, every access to the CTC must go through the step of looking up the table to compare threads. In addition, operating system must keep observe behavior of threads to enforce protection.

E. Fine-grain synchronization

Dean M. Tullsen proposed a scheme for the fine-grain synchronization inside the SMT, which means that communicating threads are internal to a single processor [1]. It is a hardware implementation of traditional software synchronization abstractions: acquire and release. They called it *Lock-box,* which is a simple hardware mechanism that enables the transfer of memory-based locks between threads on the same processor. Based upon this idea, we should think where the blocked threads, which failed acquiring the lock, should stay. Typically, it must be switched out of the SMT and stored into its address space until the operating system wakes it up. However, those threads can stay on CTC without expensive context switching operation as we discussed. I can think of this topic in a separate research since this will involve some complexity of hardware implementation for the full support for the fine-grain synchronization. Suppose that we can support that an application, which consists of multiple computation and communication threads. By providing fast communication mechanism and the synchronization scheme, the application can run asynchronously, meaning that as soon as data is available threads can start computing quickly without walking through all of the complicated interrupting procedure.

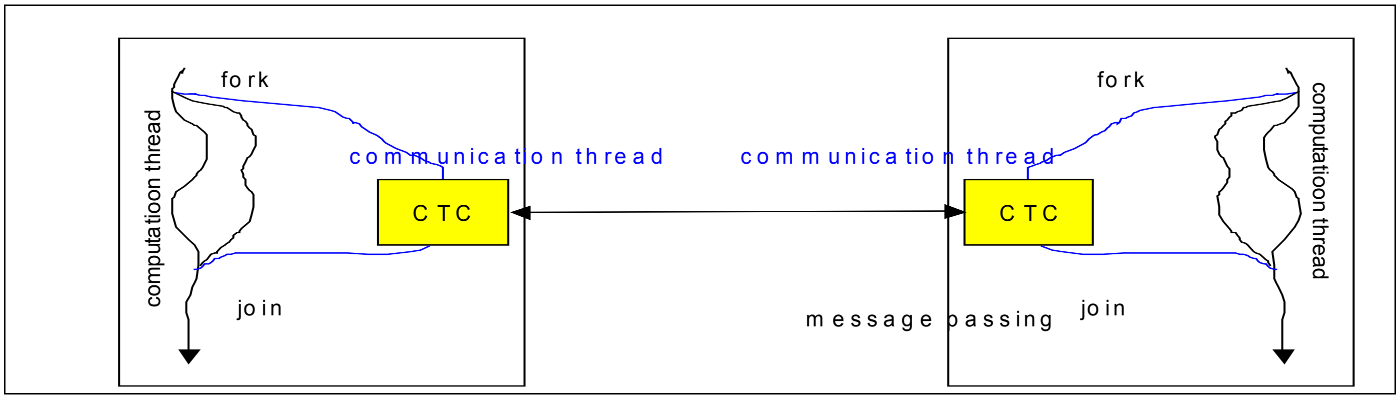

Figure 5. Different thread and different role

F. Software and Hardware cost

In terms of the cost for the software support, the operating system must handle multiple active threads, while only one thread is active in the superscalar architecture. Even operating system may not be noticed when a thread is blocked as it falls asleep in the CTC, because context switch is not involved in this scheme. However, there must be a route to inform the status of CPU such as the number of active threads and full or empty status of CTC so that the operating system smoothly handle possible problems. For instance, if all of the threads are idling, this will waste resources. Therefore, some of idling threads must be stored into the memory system, and then new threads will be able to start new jobs.

The cost for the die area to implement this scheme will be trivial, because this does not have to store all of context of a thread such as the program counter and the register file. In addition, the number of slots will be reasonably small, because the maximum number of threads, which will run on the CPU regardless of its status such as active or idling, will bound it. However, it is desirable to embed a network interface in a chip. Otherwise, an extra pin from the CPU must be dedicated to access the CTC from the outside.

G. How to increase the hit ratio on the communication thread cache?

Depending on threads running on a CPU, the ratio of communication to computation is different. If the ratio is low, the thread does not have to wait in the CTC because it will spend most of the time on computing data and threads having the high ratio need the space in the CTC quite often. If a computation thread replaced a communication thread when the cache is full, the cost of invoking the operating system for context switch will linearly increase as the frequency of communication increases. In other words, communication threads must have higher priority to stay on the cache.

IV. TIGHTLY COUPLED AND PACKET-DRIVEN SMT

I would like to propose a new architecture, called *tightly coupled and Packet-driven SMT*. It will have a SMT processor, a memory controller, a lock-box, a network interface, a SRAM L1 cache, a DRAM L2 cache, and a CTC on a chip. As shown in

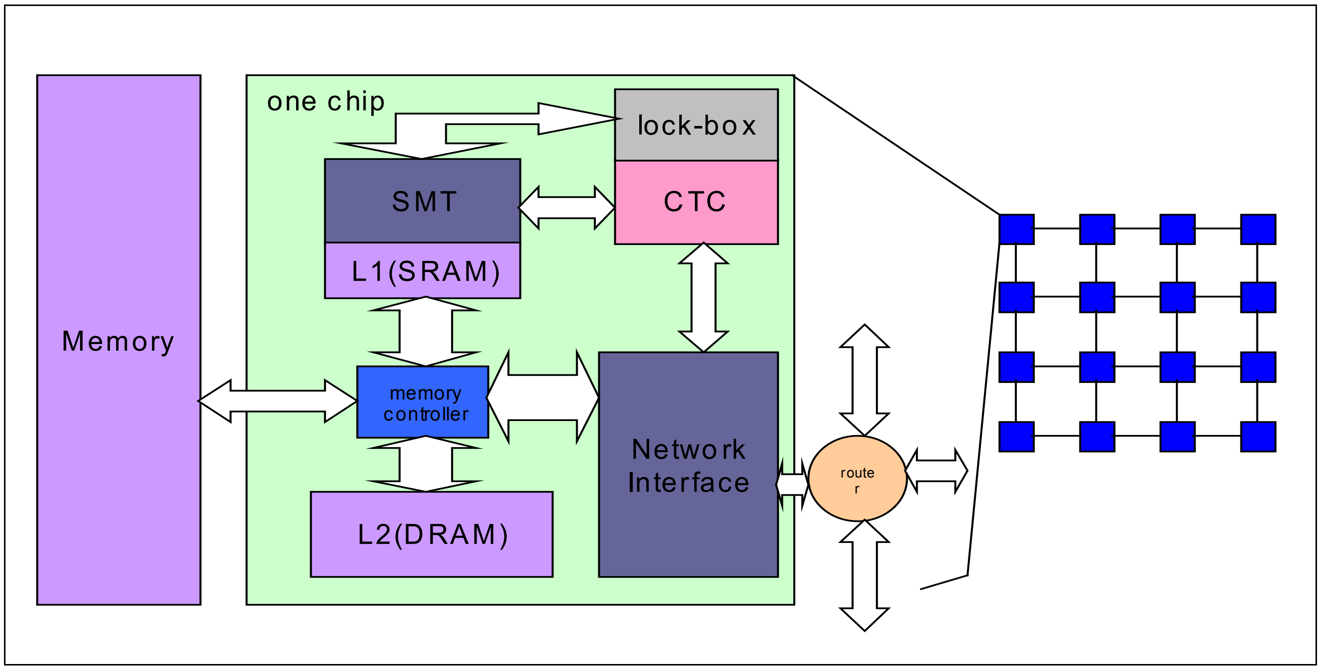

Figure 6. Tightly coupled Packet-driven SMT

Figure 6, SMT is tightly coupled with other components. L2 DRAM cache will provide enough size to satisfy memory access demands from multiple threads. Network interface must be embedded in order to have the best performance by providing fast interaction with the CTC. The lock-box will support fine-grain synchronization among threads in a chip. This architecture will present really fast communication between threads in different nodes. They communicate each other with the zero operating system intervention in the most of the cases. In the following section, two examples will show the advantages of this architecture.

V. EXAMPLES

A. DNA sequence comparison

This problem is a sequence similarity searching on protein and DNA sequence database [10]. The dynamic algorithm is simplified for the sake of easy explanation as following.

$$SM[i,j] = \max(SM[i,j-1]+gp, SM[i-1,j-1]+ss, SM-i-1,j]+gp)$$

The other elements of a matrix are calculated by finding the maximum value among the following three values: the left element plus gap penalty (gp), the upper-left element plus the score of substituting the horizontal symbol for the vertical symbol (ss), and the upper element plus the gap penalty (gp). This has strong data dependencies along the diagonal direction because values in in the left, upper-left, and upper elements must be ready to compute a new value for the current position. However, we can find anti-diagonal independencies. In other words, we can assign data in the anti-diagonal direction to nodes respectively but it will cause serious load unbalancing between processors. The processor on the corner will have small amount of data, on the other hand, a processor on the diagonal of database will stall due to heavy load. Even we balance processors with respect to the size of data in a sense that each processor has the same amount of data to process, and then the whole system will stall again due to the strong dependencies along the diagonal. However, we can generate two different threads: computation thread and

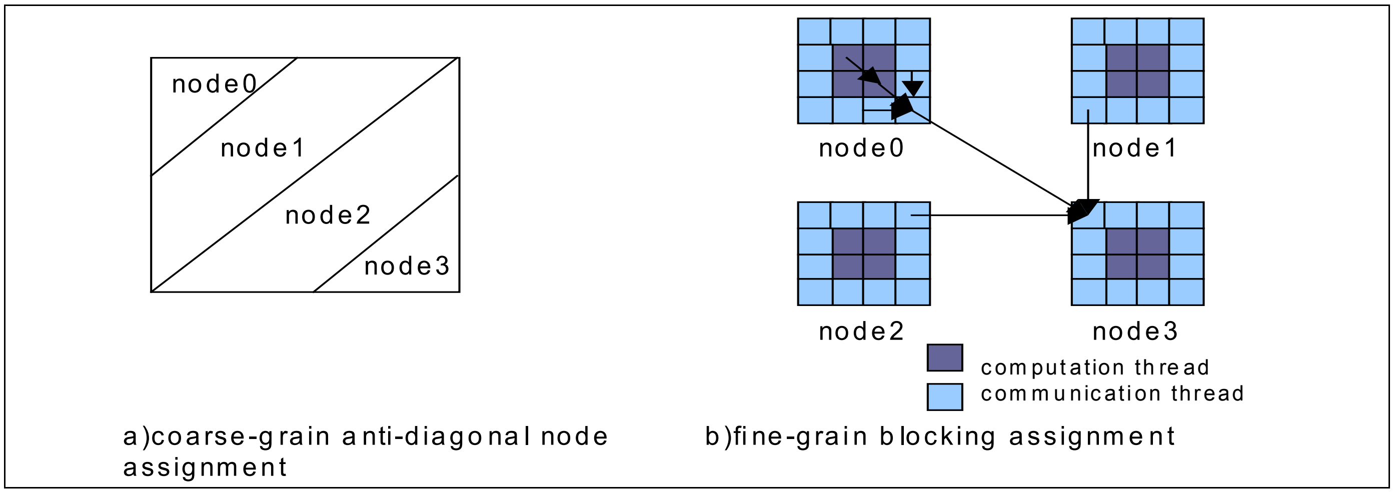

Figure 7. Mapping on SMT

Communication thread. With the fine-grain synchronization support, each thread does its role whenever its operation is required. The communication threads will send and receive data to the neighbors with the help of the CTC and the computation threads compute data as soon as they are ready. By doing this, each thread will communicate inside the processor and processors will communicate very fast. This will result in high utilization of processors since processors interact dynamically with neighbors based on the requirement of communication.

B. Active network

Packets are inherently independent each other but threads, which have to process those packets, may have *temporal locality*. In other words, if a thread processes a packet, it will be likely to process the next incoming packet. One stream of information consists of multiple packets and they require the same processing on each packet. These days, people are considering active network as a new paradigm of computer networking. Active network is a type of network supporting customized computation on the user data [11]. A packet includes a reference to identify which program should be used for processing the data in this packet. Service providers can deploy new protocols according to customers' needs, which has no effects to other protocols. Users should first download their processing routines into the required nodes. When an active packet arrives at this node, its header is examined, and the appropriate program is dispatched to operate on its contents. Therefore, the SMT processor with the CTC will be able to support those requirements very nicely. Basically, the SMT supports multiple threads on a chip and the CTC can implement fast thread invoking when the thread is in idle state in the CTC. Figure 8 describes the abstract operation

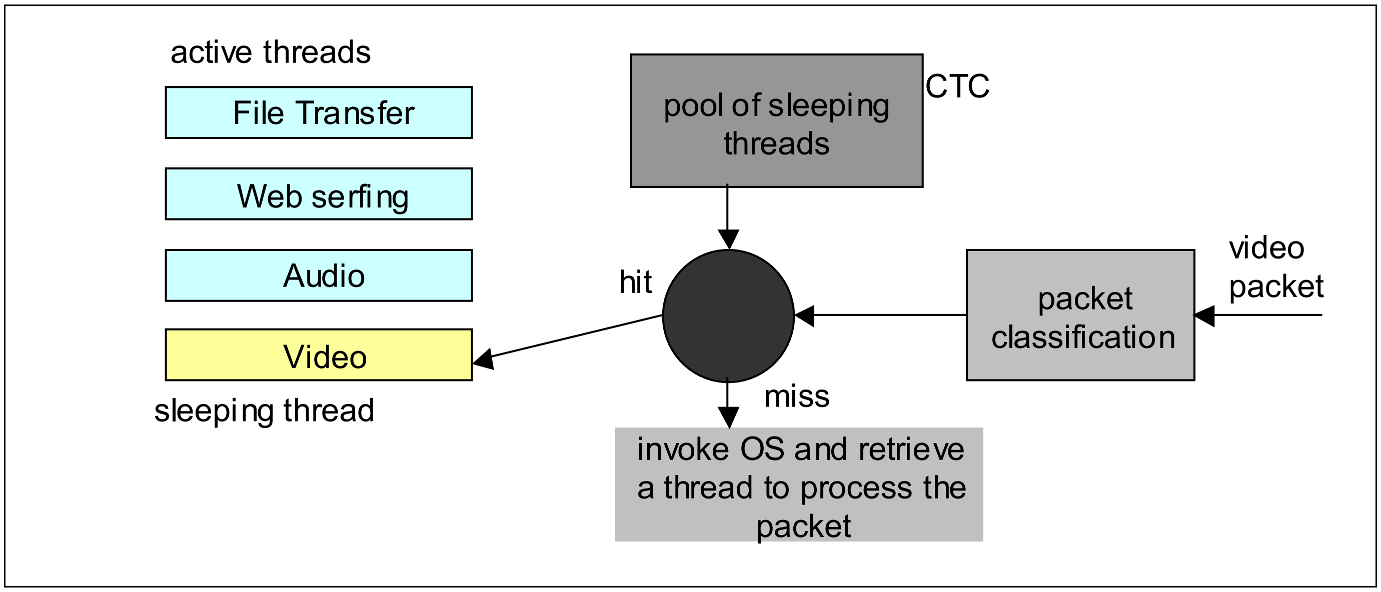

Figure 8. Different threads for different packets

of an active network processor with the SMT and the CTC. The video thread was sleeping in the CTC. A video packet arrived and the header was peeled to classify the type of the packet. Because the video thread was available, it could be reactivated without invoking the operating system.

VI. Network Modeling

In tightly coupled system as I proposed in the previous section, the latency on the physical wires must be very short. Without any operating system interruption, sending and receiving packets will smoothly continue as instructions move along the pipeline. However, the pipeline must stall when a cache miss occurs until data is fetched from the memory. Similarly, packets move along

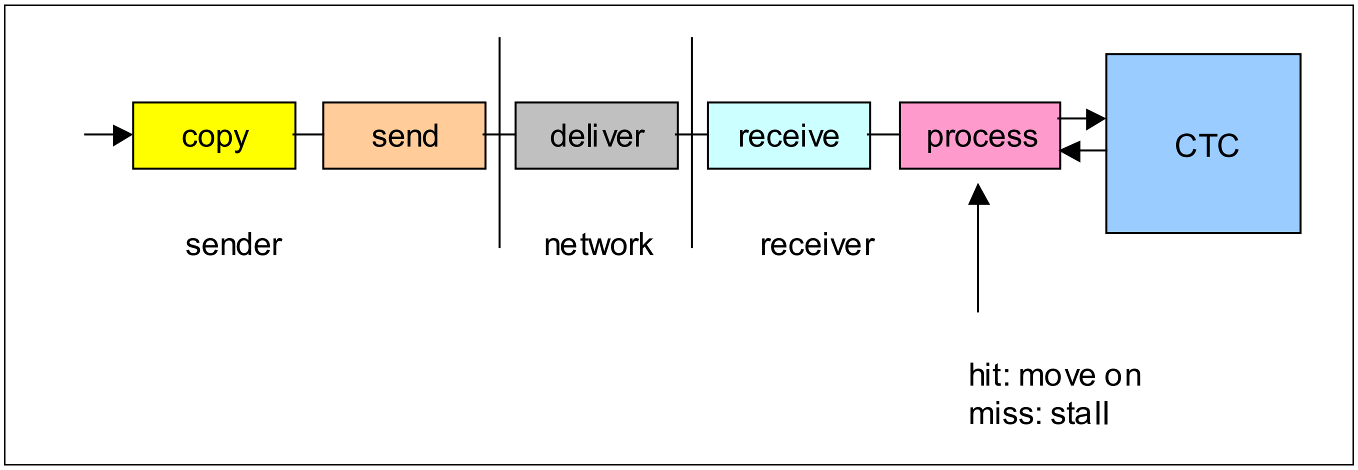

Figure 9. Pipelined network path

the network path consisting of multiple stages such as copying, sending, delivering, receiving, and processing like in Figure 9. Cache access occurs at the last stage and the pipeline stalls in the case of miss until the thread is fetched from the backing storage as done in the cache of a CPU. This modeling requires the following assumption. Each operation need the roughly same amount of time. This will be interesting measurement for this research. Finding parameters to represent this model will be very motivating. The penalty for a cache miss in CTC will be the context switching time in stead of the memory latency in a typical computer system.

VII. Future work

This idea will be very effective for the communication intensive applications and therefore exploring proper applications will be interesting. How to simulate the architecture is really an important issue in computer engineering academy. Unfortunately, there is no simulator, which has the SMT architecture and the message passing system. RSIM might be the best choice since it is based upon MIPS 10000 and has network for distributed system. Modifying it for the SMT and the message passing system will be quite a job but will be very meaningful since no SMT simulator with network system is available now. DIVA simulator is also a candidate simulator since it has the environment for the message passing system.

VIII. CONCLUSION

I proposed a novel idea, so called *communication thread cache,* to achieve a significant speedup in the tightly coupled SMT architecture. It allows us to bypass operating system interruption when a packet arrives at a destination node. Since the SMT architecture supports independent threads very well, this architecture is very proper for the active network environment. With the fine-grain synchronization support, application can be break into multiple threads based upon their roles. An network modeling is proposed, too, in order to represent this idea of caching thread.

Reference:

[1] D. Tullsen, J. Lo, S. Eggers, and H. Levy, “Supporting Fine-Grain Synchronization on a Simultaneous Multithreaded Processor,” presented at The 5th International Symposium on High Performance Computer Architecture, 1999.

[2] M. W. hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Brockman, W. Athas, A. Srivastava, V. Freeh, J. Shin, and J. Park, “Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture,” presented at SC99, 1999.

[3] C.-T. Cheng, “Architectural Support For Network-based Computing,” in *Computer Engineering Department*. Los Angeles: University of Southern California, 2000, pp. 89.

[4] S. N. Damianakis, Y. Chen, and E. W. Felten, “Reducing waiting costs in user-level communication,” presented at the 11th international parallel Processing Symposium, 1997.

[5] C. D. Cranor, R. Gopalakrishnan, and P. Z. Onufryk, “Architectural Considerations for CPU and Network Interface Integration,” in *IEEE Micro*, 2000, pp. 18-26.

[6] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X.-M. Tian, “Polling watchdog: combining polling and interrupts for efficient message handling,” presented at the 23rd annual international symposium on Computer architecture, 1996.

[7] T. v. Eicken, D. Culler, S. Goldstein, and K. Schasuer, “Active Messages: A Mechanism for Integrated Communication and Computation,” presented at The 19th International Symposium on Computer Architecture, Cold Coast, Australia, 1992.

[8] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *Computer*, pp. 79-85, 1997.

[9] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, “Simultaneous Multithreading: A Platform for Next-Generation Processors,” *IEEE Micro*, pp. 12-18, 1997.

[10] W. S. Martins, J. B. D. Cuvillo, F. J. Useche, K. B. Theobald, and G. R. Gao, “A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison,” presented at Pacific Symposium on Biocomputing, 2001.

[11] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, “A Survery of Active Network Research,” in *IEEE Communications Magazine*, 1997, pp. 80-86.