SHIFT+M: Software-Hardware Information Flow Tracking on Multi-core

Colleen Lewis Cynthia Sturton
colleenl@berkekey.edu, csturton@eecs.berkeley.edu

Abstract

We designed, implemented and analyzed three distributed protocols for information-flow tracking on a multi-core message-passing architecture. In each we used Asbestos style labels to provide protection from unauthorized communication. The protocols remove the reliance on a central repository for taint checking by adding a trusted library and hardware mechanisms at each core. We modeled the hardware and software of each protocol using Simics, a multi-core full system simulator and used micro-benchmarks to capture their respective performance with different communication patterns. We present the protocols, hardware design, and results that inform an evaluation of the three protocols.

1 Introduction and Motivation

Computers are moving towards more cores and one potential communication paradigm in the multi-core environment is user-level message passing. Message passing is well suited to a multi-core environment where network bandwidth is at a premium; one study has shown that six times as many messages are needed for shared memory cache coherence than message passing [1].

Insecure software consistently exposes users to attacks that have financial consequences for individuals and institutions [2]. Information Flow Tracking (IFT) provides a way to ensure that information flows through a system in acceptable ways, even in the presence of untrustworthy code. The goal of this project is to investigate how the security guarantees of Asbestos [2] can be distributed in a multi-core, message-passing architecture.

We describe three protocols for using Asbestos style labels to provide secure communication in a multi-core environment. The protocols have a common framework, but differ in when taint labels are sent and compared. We simulated each protocol and varied communication patterns to analyze the nature of communication that favor each protocol in terms of network load.

2 Information Flow Tracking

In each protocol we adopt Asbestos-style IFT labels in which each process has a label specifying its current state of taint, where taint is a measure of who you have communicated with. We present a simplified description of Asbestos labels to provide the necessary background to the reader.

The security goals of the system are to prevent unwanted information flow by allowing applications to set their security policy and to enforce those policies, even in the presence of compromised applications. A simple example can be given with three processes, A, B, and C. Suppose process A is allowed to communicate with process B. Once process A sends a message to process B, process B can no longer send messages to process C unless A is explicitly allowed to send messages to process C. In this way IFT can prevent the inadvertent or malicious leaking of data. To apply this example to the real-world, you can imagine that process A governs some user data, process B is a web browser and process C is a network daemon. Unless process A has given permission for its data to go out over the web, process B won't be able to send it to the network daemon.

Previous work has accomplished this by maintaining for each process a **receive label** that specifies all the categories of data that it *can* see and a **send label** that specifies all of the categories of data that it *has* seen. If process A sends process B a message, the send label of process A must be compatible with the receive label of process B for the message to be delivered. It is only compatible if process B *can* see all of the categories of data that process A *has* seen.

Labels are made up of a collection of categories. A category is represented by a 61-bit handle and a 3 bit level, which can be used to white list or black list processes. The taint labels provide the flexibility to implement military security levels or application specific policies. To create a white list, a process can create a new category and grant specific processes the right to see that data. Once a process has received a message with that data, it can no longer communicate with any process that is not allowed to receive that data. To create a black list, a process can specify a level for their category such that every process will be able to receive that data. The process that creates that category can then explicitly disallow another process, such as the network daemon, from receiving that data. Once a process receives data with this category, it will no longer be able to communicate with the network daemon or any black listed processes.

3 Previous Work

We expand upon previous work [2, 3, 4, 5] that has shown IFT to be a promising mechanism for providing security even in the face of mutually distrustful processes. Many of the current IFT implementations use a centralized directory [2, 3] in the kernel that is responsible for maintaining and enforcing the taint label constraints for all processes or objects in the system. DStar [6] is a distributed IFT system that implements HiStar-style IFT across a network.

3.1 Jif

Jif [4], a language based IFT, is an extension to the Java language that provides static analysis of information flow. It uses a labeling system to allow a program to specify and prove end to end security guarantees about the flow of its information. In contrast, in this paper we use dynamic label checking that allows programs to be written in any language, for any platform. An advantage that a static IFT like Jif provides is that it removes the covert channels inherent in dynamic checking [2, 4].

3.2 Asbestos

Asbestos [2] is an operating system that uses labels associated with each process to provide guarantees about the flow of information between processes. In Asbestos every process is able to create its own categories of taint and allow or disallow other processes from seeing taint in those categories. Although defining the security policies of the system is distributed, the mechanism for guaranteeing those policies is centralized in the kernel. In this paper we use the same labels as Asbestos, but distribute the mechanism for checking labels between many cores and move some of the work to hardware. As with the protocols presented in this paper, process communication in Asbestos is done explicitly through message passing.

3.3 HiStar

HiStar [3] builds on the Asbestos-style labels, but moves from a message passing to a shared memory paradigm. In addition to threads, objects in HiStar can have taint associated with them. Like Asbestos, the security policy is defined in a distributed fashion, but the enforcement mechanism is centralized.

3.4 DStar

DStar [6] extends the HiStar system to work with a network of mutually untrusting endpoints. It uses a combination of cryptography over the network and OS protection on the local hosts to provide the HiStar security guarantees. DStar allows any combination of IFT operating systems to be used on the different hosts. At each host there is a trusted process, the exporter, that translates locally-defined IFT labels to labels that are globally meaningful. Like DStar, the SHIFT+M design is agnostic to the type of OS running on each node. Whereas DStar relies on the OS to provide the taint mechanisms locally, we provide the taint library that all processes (including OS threads) on all nodes use to create labels.

In DStar the final checking to allow or disallow communication is done on the receive side and self-certifying certificates are used to allow exporters to authenticate themselves to other exporters. The SHIFT+M distributed protocol also compares taint at the receive side, but unlike in DStar, the sending node can trust the hardware on the receive side without any authentication.

Each DStar message must include the taint label information for the message as well as the necessary public keys to name the categories and exporters. A message in DStar with zero payload is 1348 bytes [6]. SHIFT+M protocols have inherently less overhead as the certificates are unnecessary and in this paper we explore the trade-offs of not requiring taint labels to be sent with each message.

3.5 Flume

Flume [5] is a IFT system based on the HiStar and Asbestos labels. Unlike the HiStar and Asbestos operating systems, Flume is a reference monitor that runs at the user level. It interposes on system calls to provide taint labeling guarantees. It also focuses on providing the complicated mechanisms of HiStar and Asbestos, such as port send rights and declassification privileges, in a way that makes it easier for the programmer. One weakness of Flume is that it relies on the underlying operating system to prevent a user from gaining super-user privileges.

3.6 Asbestos Policy Description Language

The policy description language [7] developed by the Asbestos group is designed to make it easier for developers to reason about and specify correct labeling policies. It provides an abstraction to Asbestos labels that allows programmers to discuss policy in terms of process communication. This work will help to make label systems a more viable option for system development in the future and further motivates the need for a well designed distributed protocol on a multi-core system.

3.4 Alewife

Alewife[8] is a multiprocessor that supports both user level message passing and distributed shared memory. Alewife's hardware support for both message passing and cache coherency provides the mechanisms we require to support message passing and a cache of communication lines. The Alewife hardware design is the basis for our design; we modify the Alewife message passing design to support information flow tracking.

4 Protocol

Below is a description of each of the three protocols presented in this paper.

4.1 Protocol 1: Send Taint Label with Every Message

The first protocol is to send labels with each outgoing message. When a node receives a message it compares the taint labels for the sending (remote) and receiving (local) process. If the two taint labels are incompatible the message is discarded, otherwise the message is delivered to the destination process.

4.2 Protocol 2: Send Taint Label with Every Message and Cache the Result

In Protocol 1, each message received requires comparing the

taint labels for the two processes, which is linear with the size of the taint label that was sent. Caching the result of this comparison will remove the overhead of the taint comparison for the case where a process sends multiple messages to the same destination process. Before directing the message to the CPU, the cache is checked to determine if the message can be delivered directly to the application or if the trusted library needs to compare the processes taint labels.

This protocol also requires a version number to be sent with each message. A version number and process ID pair represents a level of taint for a process at a particular instance in time. The process's version number is incremented every time its taint label changes. If a process's taint labels change, the cached entry will not be valid because the version number on the new message will not match the version in the cache entry.

An additional type of message, an Invalidation message, is required to allow version numbers to wrap. To handle this scenario, when the highest order bit flips, Invalidation messages are broadcast to all nodes. Invalidations must be sent twice per cycle to ensure that there is no cached data for a previous overflow of the version number. Broadcasting in a multi-core environment is not ideal, but version numbers are 32 bits and it is expected that the numbers will wrap rarely, if ever.

4.3. Protocol 3: Sending Messages without Taint Labels

Sending taint labels with every message and caching the result introduces wasted network traffic when the taint comparison has been cached. To address this weakness, Protocol 3 only sends taint labels when the receiving node requests them. In this protocol, checking whether communication is allowed and caching that information is performed on the receiving node. When sending a message, the process can assume that the message will be delivered to the destination process if and only if communication is allowed.

When a message is received the communication cache is examined to determine if communication is allowed. If it is, the message is passed to the application. If it is not, the message is pulled from the network and discarded. If there is no relevant information in the cache, the message is buffered and a Request-For-Taint message is sent to the source node. The source node will respond with a full copy of all the source process's taint labels in a Taint-Label message. When the taint label arrives at the destination node, trusted code is invoked to determine whether the communication is allowed. If the communication is allowed, the message is forwarded to the appropriate process. If communication is not allowed the message is dropped. In either case, the communication cache is updated to reflect whether communication is allowed between the source and destination processes.

The protocol also uses version numbers to ensure consistency. This is required because a process can modify its taint after sending a message but before responding to the corresponding request for taint. When this source responds to a Request-For-Taint it might send a taint label that does not reflect the state of taint at the time the original message was sent. If a process raises its taint level after sending the message, the receiving node could falsely deny communication after receiving the new, modified taint labels. Worse, if a process's taint level is lowered after sending the message the communication could be incorrectly accepted. As a solution the protocol maintains a version number to identify the state of a process's taint labels. This version number is equivalent to the version number in the second protocol and also requires the ability to broadcast invalidation messages to all nodes when the version number wraps.

4.5 Description of Alternative Protocols

The following designs are not evaluated in our simulation. A description of each design and rationale for exclusion are provided below.

4.5.1 Communication Cache on the Send Side

An alternative protocol would be to cache whether communication is allowed on the send side instead of on the receive side. In this protocol, the sender would check its cache and only send a message if communication is allowed. In the case where there is no cached information, the sender would request the taint label from the receiver and then do the comparison before sending out the message. This protocol has the seeming advantage that the side that initiates the communication has to do most of the work (checking cache, buffering message, and comparing the taint labels). The problem with this protocol is that the comparison of taint does not occur at the same time that communication takes place, which is the point when the message reaches the receiving node. If the receiving node has changed its taint label between the time when it sends out its taint in response to a request and the time when the message arrives, it cannot trust that it is able to accept the message and so must drop it. It can then tell the sender it must re-initiate the handshake and resend the message. This cycle can continue if the receiving side changes its taint labels again.

4.5.2 Check Taint Labels in Hardware

It might seem advantageous to compare taint labels in hardware. However, this is not a viable option, given the size and potential number of taint labels. A single taint category is 64 bits and a taint label might include hundreds or even thousands of categories. In OKWS [2], the Asbestos web server each user session requires two categories to fully isolate it from other sessions. The few processes that communicate with every user session will have two times the number of categories in their receive label as active sessions.

This illustrates an extreme case, but even in the more general case, it is expensive to compare taint labels in hardware. If instead, we perform the comparison in software, we can use a hash table to more efficiently compare taint labels when the send label is much shorter than the receive label.

5 Hardware Design

The SHIFT+M architecture is designed to support message passing IFT. The three protocols proposed in this paper require the ability to send a message, receive a message, and invoke the trusted library to manage taint labels. In addition to these basic functions, Protocol 2 and Protocol 3 require a cache of communication lines. SHIFT+M is largely based on the MIT Alewife architecture which supports quick user level messaging, efficient interrupts to trusted code libraries, and memory to memory or register to register data transfer. These are the key components necessary to efficiently manage the taint labels in a message passing environment.

5.1 The MIT Alewife Architecture

SHIFT+M uses the Alewife architecture with some modifications made for taint label management. We present here an overview of the Alewife machine, focusing only on those aspects which directly affect message passing and cache maintenance [9].

Alewife can support both user-level messaging and shared memory with a sequentially consistent coherent cache protocol. The design takes advantage of the fact that cache coherency messages existing in shared memory multiprocessors already provide the hardware necessary to provide user-level messaging. In order to share hardware, all messages, cache coherent or user level, have the same header format and structure. In addition to exposing the message passing substrate to the user, Alewife provides direct register to register transfer and DMA assisted memory to memory transfer for quick user level messaging. The Sparcle processor used by Alewife provides additional contexts, one of which is reserved to preclude the need to save and restore register state when vectoring the CPU to the appropriate interrupt handler on message arrival. The Communication and Memory Management Unit (CMMU) manages the cache and provides user access to the interconnect.

5.1.1 Messages

In Alewife, the user can directly access the processor interconnect via loads and stores to the input and output queues. The processor creates a message by writing to the output descriptor queue. It can write either payload data or address-length pairs describing the payload's location in memory (the payload can be scattered). In the latter case, the processor will invoke the DMA engine to marshal up the message and place it on the outgoing network queue. All incoming messages that are not cache coherency- related (as determined by the message header) interrupt the processor by causing it to vector to the appropriate interrupt handler. When such a message arrives, it is placed on the input descriptor queue while the processor simultaneously flushes its pipeline and vectors to the interrupt handler. The processor can then inspect the first 8 64-bit words of the message from the input descriptor queue and choose to either discard the message, read the contents of the message from the input descriptor queue or instruct the DMA to read the contents of the message and store it into memory for later consumption. Coherence protocol messages are handled directly by hardware in the CMMU.

5.1.2 Cache

The CMMU manages the cache, pulling protocol-related messages off the input queue and putting outgoing protocol messages on the output queue. The Alewife design uses the LimitLESS [10] cache directory design which extends the cache directory into memory if many nodes are sharing the same cache line. This feature requires that software be able to generate cache protocol messages. However, the hardware must ensure that user-level code cannot generate protocol messages. This is done by checking the header information of outgoing messages and discarding any user-level messages that have an illegal opcode. The ability for software to generate protocol messages is a feature that is only possible because of the unified interface to the interconnect and, as we will see in the next section, it is a feature that is crucial to the design of SHIFT+M.

5.2 SHIFT+M Architecture

We discuss here the modifications we make to the Alewife design in order to support Information Flow Tracking.

5.2.2 Communication Cache

Protocols 2 and 3 in SHIFT+M require a cache be kept of communication lines. This replaces the shared memory cache in Alewife. The cache line holds the local process ID, the remote process ID, the taint version number for the remote process and whether or not the communication is allowed.

For our initial design we have chosen a large cache of 64KB or 256 cache lines. We propose a cache that is 4-way associative with the 4 low order bits from the source and 4 low order bits from the destination used as the index. The 60 high order bits from both the source and destination process ID are used as the tag. We want to use a combination of the source and destination process ID as the index to avoid having all of a particular process's cache lines conflict.

5.2.3 Process ID and Version Number

When messages arrive the header is inspected. If it is a user-level message and the communication is allowed, the processor will vector to an interrupt handler which can inspect the destination process ID in the header of the message in order to vector to the correct process' interrupt

handler. Otherwise, the interrupt handler will vector to the correct kernel-level label management code based upon the type of message indicated in the message's major opcode.

Protocol 2 and 3 require that a version number is included in each message. For Protocol 2, this version number can accompany the label data and be added to user messages with the same mechanism that is used to add the label data. Protocol 3 avoids accessing memory on a message send. To achieve this, the process ID and version number for the current process are stored in privileged registers. These register values, which cannot be modified by user level code, are written to every message sent. The user fills in the entire header information, but any value the application writes to the process ID or version number fields will be overwritten by the values from the HW registers. This requires that the operating system code for assigning and maintaining process IDs be trusted.

Figure 1 illustrates the additional hardware necessary to support this scheme. The hardware is insignificant compared to the existing cost of the communication cache. Therefore Protocol 3 is not significantly more expensive in hardware costs than Protocol 2.

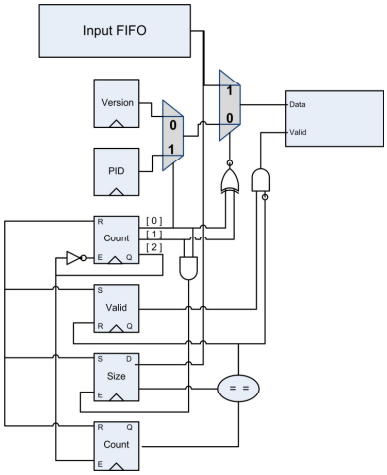

Figure 1: Writing process ID and version numbers.

5.2.4 Message Headers

Since the same queues are used for both cache related traffic and user messages, all messages must have a uniform header [9]. SHIFT+M abides by this restriction, but modifies the header to include the necessary data. Figure 2, Figure 3, and Figure 4 describe the message headers for each protocol. Table 1 lists the major message types used by each of the three SHIFT+M protocols. The high order bit of the message type is used to determine if the message should be directed to the communication cache (for User-Message, Taint-Labels, Grant-Rights, and Invalidate).The size signifies the number of 64 bit words in the payload. A size of zero means that there is no payload accompanying the header. This is the case for the Request-For-Taint message and the Invalidation message. Although there are only 3 bits used for the type the field is padded to 32 bits to simplify the hardware.

Figure 2: protocol 1 message header

Figure 3: protocol 2 message header

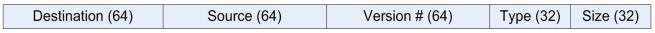

Figure 4: protocol 3 message header

Protocol 1	Protocol 2	Protocol 3
User-Message (001) Grant-Rights (010)	User-Message (001) Grant-Rights (010) Invalidate (011)	User-Message (001) Grant-Rights (010) Invalidate (011) Taint-Labels (000) Request-For-Taint (100)

Table 1: Message Types

6 Handling Message Types

For each of the three proposed protocols we describe the path a message takes through the hardware as it passes between the interconnect and the node. Depending on the message type, it may pass directly to be handled by the processor, it may be handled exclusively by the communication cache, or it may require participation by both the cache and the processor. The discussion here motivates our simulation design and provides the rationale behind our chosen simulation parameters. Figure 5 illustrates the dataflow for Protocol 1. Figure 6 illustrates the dataflow for Protocols 2 and 3.

6.1 USER-MESSAGE

The User-Message is used by all three protocols. This is the message that one application uses to communicate with another.

6.1.1 Protocol 1

If an application chooses to send a message to another application, it can do so directly using the Alewife messaging interface. The difference here is that once the user has written either its message or the address-length pairs describing its message to the output descriptor queue and launched the message, the processor will trap to the kernel level message handler which appends the address-length pairs describing the location of sending process's taint labels. The DMA engine will then append this information to the outgoing user level message. This information is appended to every message and the user has no chance to modify it before the message is put on the network.

When a message of type User-Message is received, the processor is vectored to the kernel level message handler which instructs the DMA to store the message into memory. The message handler then reads the taint information of the destination process from memory and does a label comparison. A pair-wise comparison is done for every category in the sending process's label to the corresponding category in the destination process's receive label. Additional

categories in the receive label do not need to be checked. For this reason, send labels are always stored in memory as an unordered array while receive labels are always stored as a hash table. If the communication is allowed, the message is copied to user space, control is switched to the receiving process's message handler and the message is delivered. If the communication is not allowed the message is discarded and its memory is freed.

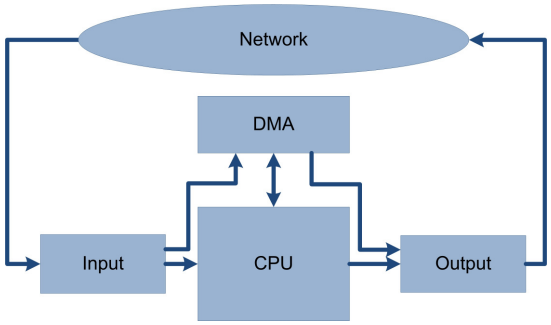

Figure 5: Dataflow for Protocol 1.

6.1.2 Protocol 2

Sending a User-Message is the same as in Protocol 1.

An incoming message on the input descriptor queue of type User-Message will prompt the CMMU to check the cache for a valid entry with matching source PID, destination PID, source round number. If the entry is found and it indicates that communication is not allowed, the hardware will discard the message. If the entry is found and communication is allowed, the processor is interrupted and vectored to the message handler for the destination process. If there is no corresponding entry in the cache then the processor is interrupted and vectored to the kernel level label comparator which reads the taint information of the destination process from memory and does a label comparison. The message is either discarded or delivered to the user as in Protocol 1.

6.1.3 Protocol 3

In Protocol 3 sending a User-Message is the same as sending a user-level message in the original Alewife design. A user writes either the message data or the address-length pairs locating that data to the output descriptor queue and then calls launch to send the message.

Receiving a User-Message in Protocol 3 only differs from Protocol 2 in what happens when there is no entry in the taint-handshake cache. For Protocol 3, the processor is vectored to the trusted library code for message buffering which instructs the DMA to buffer the message into memory. The communication cache will create the Request-For-Taint message and put it on the network output queue to be sent to the original source node.

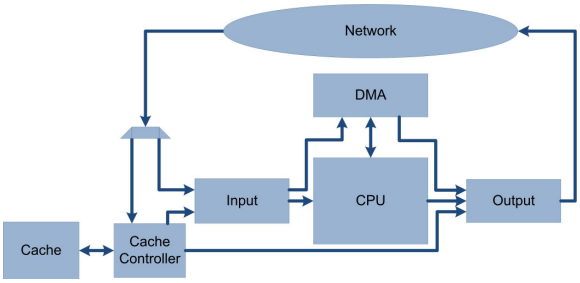

Figure 6: Dataflow for Protocols 2 & 3.

6.2 GRANT-RIGHTS

6.2.1 Protocol 1

Applications can choose to grant receive access in categories that they own to other processes by making a SHIFT+M library call. The trusted library will confirm that the process has ownership of that category and then will generate a message granting those rights to the receive process. In the first protocol, the Grant-Rights message needs to be appended with the taint labels of the sending process just as for the User-Message.

When a Grant-Rights message arrives on the input descriptor queue, the processor will be interrupted and vectored to the kernel-level message handler so the receiving process's taint labels can be updated. Before the update can take place, the message handler will first confirm that the process has the ability to receive from the sender. This process is the same as in the case of the User-Message. Since receive labels are stored in memory as a hash table, adding new categories happens in constant time.

6.2.2 Protocol 2

Sending a Grant-Rights message is the same as in Protocol 1.

Receiving a Grant-Rights message is similar to receiving a User-Message except that instead of vectoring the process to the user-level message handler, the kernel-level label management code is invoked so that the new receive label can be updated. The rest of the protocol is the same, including the cache look-up, label comparison if necessary, and possibly discarding the message.

6.2.3 Protocol 3

Sending a Grant-Rights message involves a SHIFT+M library call just as in Protocol 1 and 2. The difference is that in Protocol 3, the Grant-Rights message can be delivered without appending the taint labels of the sending process.

Receiving a Grant-Rights message is similar to receiving a User-Message message except that if the cache shows that the message can be received, the processor is vectored to the label management code so the labels can be updated. If the cache shows that receiving messages from the source process is forbidden, the Grant-Rights message is dropped. If the cache does not have valid information, the Grant-Rights message is buffered and a Request-For-Taint message is sent to the source process.

6.3 INVALIDATE

6.3.1 Protocol 1

Invalidate messages do not exist in Protocol 1 as the protocol does not use a communication cache.

6.3.2 Protocol 2

The label management code sends an Invalidate message after receiving (and allowing) a Grant-Rights message. It fills in the source PID to reflect the process whose labels were just updated. The Invalidate message is put on the network queue as usual, but it is addressed to the local node. The label management code can also generate an Invalidate message when a process's taint version changes and the high order bit of the version number flips. In this case the Invalidate message is sent to a broadcast address that will be delivered to every node.

Allowing the label management code to generate cache protocol messages is a direct application of the LimitLESS design [10]. LimitLESS, the cache directory protocol in Alewife, allows the cache directory to be extended into memory by software if needed and the software to generate cache protocol messages when required. The user is prevented from generating fraudulent protocol messages by hardware which inspects the headers of outgoing messages and drops any user messages with illegal headers.

When the Invalidate message appears on the node's input queue, it is handled by the CMMU which invalidates all the communication cache lines for the appropriate PID.

6.3.3 Protocol 3

Invalidates work the same way for Protocol 3 as they do for Protocol 2.

6.4 TAINT

6.4.1 Protocol 1

Taint information is appended to every outgoing User-Message by the kernel level message handler. See section 6.1.1 for a full explanation.

6.4.2 Protocol 2

Taint information is appended to every outgoing User-Message by the kernel level message handler. See section 6.1.2 for a full explanation.

6.4.3 Protocol 3

In this protocol a Taint message is created and sent in response to a Request-For-Taint message. See section 6.5.3 for a full explanation.

When a Taint message is received, the CMMU first checks the communication cache to see if there is already a valid entry for the source PID, source round number, destination PID given in the Taint label. If there is an entry then no further action is taken. This might happen if the local process received multiple User-Messages and for each one sent out a Request-For-Taint. As soon as the first reply Taint message is received, all the waiting messages will be pulled from the buffer and delivered to the process (or not, if communication was not allowed).

If there is not valid entry in the communication cache, the processor is interrupted and vectored to the taint management library code. That code instructs the DMA to store in the entire Taint message in memory and then does a pairwise comparison of taint labels, as described in section 6.1.1, between the destination process's taint labels and the labels in the Taint message. If the comparison shows that communication is okay, the taint management code copies any messages destined for the destination PID from the waiting messages buffer into user space and the destination process's message handler is invoked. If comparison shows that communication is not allowed, the messages in the waiting messages buffer are marked as invalid and the space is freed. In either case, the CMMU is instructed to update the communication cache with the new information.

6.5 REQUEST TAINT

6.5.1 Protocol 1

There is never a need to request taint in Protocol 1 since taint information is appended to every outgoing message.

6.5.2 Protocol 2

There is never a need to request taint in Protocol 2 since taint information is appended to every outgoing message.

6.5.3 Protocol 3

When a Request-Taint message is received the processor is interrupted and vectored to the taint management library code. The software can read the header of the Request-Taint message from the input descriptor queue to see for which PID and round number it should look up the taint labels. The taint management code will put the headers of the new Taint message on the output descriptor queue, followed by the address-length pairs describing the locations of the complete send label for the particular process. This is the usual method of generating a message; once the software is ready, it will call launch and the DMA will copy the message data from memory and put the full message on the network output queue.

7 Simulation

We used Simics, a full system multi-core simulator from Virtutech [11] that provides cycle accurate simulation for a number of multi-core architectures. None of the architectures provide message passing, so we started with x86-440 and implemented message passing functionality. We have generated modules that we plan to contribute to existing repositories of Simics modules. Our simulation includes three modules: MP_CommunicationCache, MP_MessagePassing and MP_Network. MP_CommunicationCache and MP_MessagePassing each take input parameters to specify that no protocol, Protocol 1, Protocol 2 or Protocol 3 should be used. The three Simics modules total 2900 lines of commented code.

7.1 Modules

7.1.1 MP_MessagePassing

There is one instance of MP_MessagePassing per core. This module provides the basic user-level message passing functionality of Alewife. The user send and receive instructions interface with this module. In order to create the send and receive functionality, we created a library for send and receive. When an application wants to send or receive a message it can use one of these library functions. The library uses inline assembly to emulate ISA send and receive instructions. For a send, it writes the message contents to a particular location in memory and then writes the location and length of the message to a set of reserved registers. The MP_MessagePassing module will observe these register values immediately and copy the message from the application's memory space into the simulation space. Once a message has been brought into the simulation space, it can be passed on to MP_Network.

The application polls to receive messages from the network. Once the entire message contents are available, MP_Network writes the process ID of the destination process to the memory location.

We use this module to model the costs to the CPU of the different protocols. It is from this module that we stall the CPU when the protocol requires the CPU be interrupted and vectored to a SHIFT+M library. This might happen when doing the taint comparison after receiving the Taint message or instructing the DMA to buffer a message while waiting for a Taint message. We discuss the different parameters used in our protocol in section 7.3.

7.1.2 MP_CommunicationCache

There is one instance of an MP_Taint_Hardware per core that works in conjunction with MP_MessagePassing to handle messages coming in off the network and is meant to module the functionality of the CMMU in Alewife. It examines the header of the incoming message and takes the appropriate action as described in Section 6. This may include checking the cache, creating a new Request-For-Taint message, or invalidating cache lines.

7.1.3 MP_Network

There is a single MP_Network module that connects all the MP_MessagePassing/MP_CommunicationCache nodes. The MP_Network routes based upon the high order bits of the destination process ID. We assume the mesh network used by Alewife, but abstract out the network routing. The processes register their core location with the MP_Network module out of band with a special Register message type. The MP_Network can then deliver messages to the correct core using the registered information it has for each core. Processes are responsible for addresses their messages to a know PID.

MP_Network is responsible for tracking and modeling network costs. Latency and network congestion is roughly simulated by delaying delivery of a message by a number of cycles that depends on the length of the message and the number of messages in the network plus a constant propagation delay. Although the network only approximates true per-message latency, it is included to ensure that Protocol 3 is duly penalized every time it has to perform the three-way handshake. We use numbers that err on the side of too high a latency.

7.2 SHIFT+M Libary

The SHIFT+M libraries are responsible for all the taint label management such as granting new categories or comparing two taint labels for compatibility. In our current simulation the SHIFT+M libraries run inside MP_CommunicationCache. Since computation inside the hardware module does not cost any simulated CPU time, we ran the library code once outside of the Simics environment in order to time the jobs. We then use those numbers to know how long to stall the CPU when performing the library functions.

7.3 Parameters

We present here the parameters we are using in our simulations. Many of the costs associated with the three protocols proposed in this paper are heavily workload-dependent: they depend largely on the size of the taint labels and the number of processes a single process is communicating with. We designed our analysis to highlight the costs and benefits of using a cache and a three-way initialization handshake. We avoided running simulations with our synthetic benchmarks that told us no more than the information we supplied in our parameters. Once we have ported the Asbestos OKWS web server to our simulated environment and we have a realistic workload, these parameters will give us a fair idea of the costs of bringing IFT to the multi-core environment and the benefits of moving part of the work involved with IFT to hardware.

7.3.1 Cache Size

The cache size would ideally be as large as the total number of communication partners of all the processes on that node. Until we have realistic workloads, we can not empirically determine a reasonable working set and corresponding cache size. In order to test the three protocols with a cache that is not over-provisioned we have simulated a 16 entry cache that is 2-way associative.

7.3.2 Context Switching

The Alewife architecture supports quick interrupts by vectoring to the appropriate message handler. We assume the 11 cycle cost of a context switch in Alewife [10].

7.3.4 Taint Comparison

The CPU cost of comparing taint labels depends linearly on the number of categories in the send label. In order to make the lookup of receive categories quick, the send taint categories are stored in an unordered array and receive categories are stored in a hash map.

7.3.5 User Message Size

The average payload of a user message can vary greatly with the application.

7.3 Synthetic Benchmarks

We have written synthetic applications to compare the three protocols in terms of network traffic as affected by communication patterns.

The two aspects of the protocol we wish to test are the cache of communication lines and the potential savings of an initial handshake as opposed to sending taint with each message. We have designed the following tests to that end.

8. Results

8.1 Increasing Allowed Communication

In the following synthetic benchmark we investigate the impact of traffic patterns on network load. We varied the number of processes running on each of two cores and tracked the total network traffic required to send 100 user messages. In the benchmark, each process communicates with one other process in a send, wait, receive, loop and has a random amount of stall time between receiving and sending a message to simulate computation. Each process is initialized with three categories of taint before the measurements are made to focus on the network load from taint tracking rather than the cost of creating new categories. While the system provides secure information flow, this benchmark models the expected behavior of processes that attempt only allowed channels of communication.

We ran the benchmark with four conditions: without taint tracking and with each of Protocols 1 through 3. The graph below (Figure 7) shows the total network load for 100 user messages in each protocol. This load represents the necessary headers and 8 - 64 bit words of payload with each user message. As appropriate, messages including taint were sent with 3 - 64 bit taint categories.

Only Protocol 3 has the possibility of a variable network load for a given number and size of messages. However, the results of Protocol 1 and communication without taint tracking provides information regarding the relative cost of information flow tracking in the various protocols. The network load of the non-information flow tracking instance shows a base line cost of real communication.
Protocols 1 and 2 have an equivalent impact on the network load, which is less than the network load of Protocol 3 for high levels of connectivity. At low levels of connectivity Protocol 3 has less impact on network traffic because a high cache-hit rate allows handshakes to happen infrequently. The simulation used a 2-way, 16 entry cache. The network load of Protocol 3 is directly impacted by the size of this scaled down version of the cache. The cache hit rate, as measured by total number of requests for taint generated, is shown below (Figure 5) to demonstrate the degree to which Protocol 3 is affected by cache size.

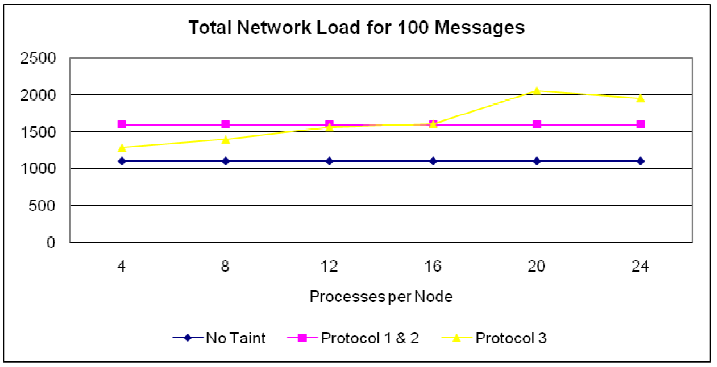

Figure 7: total network load for sending 100 user messages

9 Future Work

9.1 Quick Deny

Our current implementation assumes the common case is that attempted communication will be allowed. In the case that denying communication is more likely, an additional hardware mechanism could be added to avoid the costly comparison of taint labels. If the source process has more high level send categories than the destination has high level receive capabilities, there is no way for their labels to be compatible. Meta data about the number of high level capabilities for each process could be calculated whenever a process's label changes. This taint meta data could be sent with each message in Protocols 2 and 3, along with the process ID and version number. On the receive side the meta data tag for the sending and receiving process could be checked before beginning the full check to check for the possibility of quickly denying at a much lower cost than invoking the CPU to do the comparison.

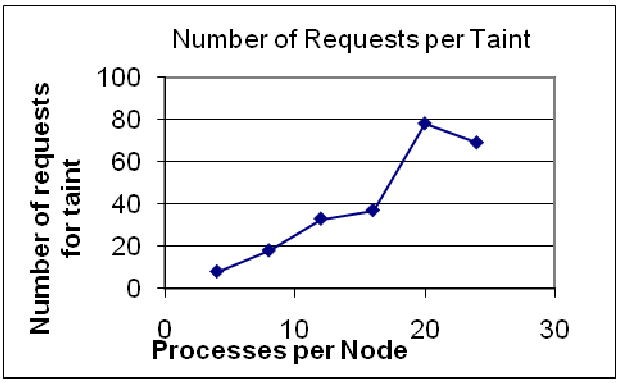

Figure 8: number of requests for taint for protocol 3

9.2 Pending Taint Request

In our current protocol, if process A sends two User-Messages to process B, process B may generate two Request-For-Taint messages before receiving the requested Taint-Label from the sender. This causes wasted Requests-For-Taint traffic, wasted Taint-Label marshaling and wasted Taint-Label traffic. The duplication of work could be avoided by including a pending value in our cache. When taint is requested, the cache is updated with a valid cache entry with the pending bit set to 1.

9.3 Port the Asbestos Web Server to our Simulated Environment

Port the OKWS to the multi-core environment and instrument it to use the taint libraries and hardware mechanisms on our simulated message passing architecture. The Asbestos OKWS web server is written in the message passing style and would provide a realistic workload for our protocol. Our expectation is that the traffic patterns in the Asbestos web server will justify the costs of a communication cache and possibly also the overhead of the initial three-way handshake.

9.4 Allow shared memory in addition to user level messaging.

A message passing architecture does not preclude the existence of shared memory on the same machine. Recent work [12] has focused on how the use of HiStar protection can be optimized for use in a shared memory environment. Future collaboration can integrate the two on an architecture such as Alewife that supports both shared memory and message passing.

10 Conclusion

We presented three possible protocols for ensuring secure communication in a multi-core system. In the worst case, Protocol 2 has equivalent software costs as Protocol 1, with the additional hardware complexity to maintain a cache. This cost can be counteracted by the possibility of avoiding unnecessary taint comparisons. As in Protocol 3, it is advantageous when communicate involves multiple messages. Although for Protocols 2 and 3 to achieve this advantage it is important that applications avoid changing their taint level after beginning communication because modifying their taint will invalidate any successfully established handshakes.

Protocol 3 The initial handshake does add latency, but the cost may be small compared to the latency of sending all the taint categories and doing a full comparison, so there may be little effect on total throughput.

Protocols 1 and 2, rely on processes having few taint labels. Although, in the example of the Asbestos OKWS web server, the cost of sending taint labels even once would be quite costly. In any system where taint is associated with threads rather than with data, there must be a few threads that are trusted to guard data stores. The Asbestos OKWS web server uses this model to isolate and protect disparate user data in the back-end database. Given this pattern of design, where a process may have thousands of taint categories, it is imperative that taint labels are sent as infrequently as possible. Protocol 3 achieves the least frequent transmission of taint labels.

11 References

[1] Chong, F., Barua, R., Dahlgren, F., Kubiatowicz, J., and Agarwal, A. 1998. The Sensitivity of Communication Mechanisms to Bandwidth and Latency. In *Proceedings of the 4th international Symposium on High-Performance Computer Architecture* (January 31 - February 04, 1998). HPCA. IEEE Computer Society, Washington, DC, 37.

[2] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th SOSP*, pages 17-30, October 2005.

[3] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, pages 263-278, November 2006.

[4] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410_442, October 2000.

[5] "Information Flow Control for Standard OS Abstractions." Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, Robert Morris. In *Proceedings of the 21st Symposium on Operating Systems Principles* (SOSP 2007), Stevenson, Washington, USA, October 2007.

[6] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In *Proceedings of the Fifth Symposium Networked Systems Design and Implementation (NSDI 2008)*, San Francisco, CA, pages 293-308, April 2008.

[7] Petros Efstathopoulos and Eddie Kohler. Manageable Fine-Grained Information Flow. In *Proceedings of the* 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, Glasgow, UK, April 2008.

[8] Anant Agarwal , Ricardo Bianchini, David Chaiken, Frederic T. Chong, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine**.** *In IEEE Proceedings, 1999*

[9] Kubiatowicz, J. and Agarwal, A. 1993. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 7th international Conference on Supercomputing* (Tokyo, Japan, July 19 - 23, 1993). ICS '93. ACM, New York, NY, 195-206. DOI= http://doi.acm.org/10.1145/165939.165970

[10] LimitLess Directories: A Scalable Cache Coherence Scheme, David Chaiken, John Kubiatowicz, and Anant Agarwal, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224-234, April 1991

[11] http://www.virtutech.com/

[12] Bird, S., McGrogan, D. Hardware Support for Enforcing Information Flow Control on ManyCore Systems. Class Project, CS262, CS258, Spring 2008.