ONStor
Storage Solutions for the Content Generation

Project Proposal

Prepared for: Brian Stark, V.P. of Cool
Prepared by: Bill Fisher and Andrew Sharp

October 27, 2008

Onstor 254 East Hacienda Ave. Campbell, CA 95008 **T** 408.376.3116 **F** 408.963.2409 andy.sharp@onstor.com

Executive Summary

Replace eee on the Cougar platform's TXRX Sibyte 1480 CPU node with Linux, and convert the ACPU software (NFS and CIFS protocol stacks) to run as a thread under Linux. The project will be called TuxRx ("Tucks-Ar-Ecks"), a combination of the TXRX name and Tux, the Linux penguin mascot.

Benefits Summary

- Increased Performance (utilize all 4 CPU cores on the Sibyte 1480).
- Multithreaded IPv6 support; community maintained.
- Multithreaded IPv4; community maintained.
- FTP, HTTP and other application protocols.
- QoS capabilities including bandwidth control.
- Compression and Encryption for DMIP.
- 10-Gigabit drivers which we would otherwise have to write and maintain ourselves, and would be thrown away when this phase of System-X is implemented.
- iSCSI protocol support (1/2 of the iSCSI picture).
- Big step towards realizing System-X: together with the other System-X work on the agenda for Kegg, this will comprise from 60 to 70 percent of the total System-X implementation work.

Objective

1. To obtain a more updated TCP/IP stack which provides multithreaded support for both IPv4 and IPv6. An adoption of the Linux TCP/IP protocol stack allows obtaining IPv4, IPv6, bonding driver improvements and 10 Gigabit Ethernet support easily. In addition, the standard IP based protocols such as FTP, HTTP and others come without major development effort, including security and functional maintenance.

2. The support of 10 Gigabit Ethernet device drivers via deploying standard vendor Linux NAPI device drivers for all the commonly available 10-GigE hardware..

3. The adoption of Linux on the SiByte 1480 TXRX is the second step in the migration to System-X. Using the existing Cougar hardware platform allows us to use one of the two Sibyte 1480s to execute Linux with the other one supporting the FP functions unchanged. This plan allows a transition of the NCPU (transport protocols) and ACPU (NFS/CIFS) functions to Linux with a minimum of disruption to the rest of the system.

Scope

The current feeling is that this development work will culminate in the March 2009 time frame. Plans call for taking the task summary below and turning it into a development plan with detailed scoping numbers in the next two weeks (Nov 4, 2008).

General Task Overview

In the sections below, the identified tasks are listed in approximate chronological development order. There are a number of things that must be done first before the NFS/CIFS functionality can be tested. Hence there's room for more schedule parallelism depending on the resource allocation.

1. Port stock Linux 2.6.{26,27} kernel onto the one Sibyte socket

   The goal is to "port" a stock Linux 2.6.{26,27} kernel onto the TXRX node, aka the Sibyte 1480 socket supporting 4 processor cores. This will be a very stripped down kernel supporting the minimum number of device drivers, file systems and user functions. Since a Linux 2.6.22 kernel has been ported to the SSC Sibyte 1125 processor as part of the 4.0 release, it's envisioned this is a very short and straight forward task.

2. Loading Linux Kernel Modules

   This task concerns loading the Linux kernel modules after the SSC has loaded Linux on the 1480.

The SSC PROM loads all three (SSC Linux, TXRX, FP) images from the Compact Flash into memory and then notifies the various processors to start execution.

In the current software, after the Sibyte 1480's images are loaded, the Embedded Eagle Executive (EEE) requires no further code loading functions. The TuXRX Linux Kernel will need to load modules itself after it has been loaded into memory by the SSC PROM. This is a challenge because the 1480's don't have direct access to the CF cards from which they would need to get the modules.

One approach is to store these kernel modules on the CF. In order to access the CF, a communication channel is needed between TuXRX Linux and SSC Linux, to allow modules to be read from CF. If we extend the management-bus driver running on the SSC to pass TCP/IP packets, this "point-to-point" link could be used to access the CF via NFS.

Using NFS as the upper level protocol running over this link, the CF file system can be mounted onto the NPCU. Since the module we are loading would be the OnStor NFS/CIFS module, we have the classic chicken-and-the-egg problem during the module loading phase. This has the side-effect of requiring the Linux NFS client code to support the file operations. This would only be required for development configurations: a production version would utilize an initial RAM disk loaded by the PROM along with the kernel.

3. Support Shared Memory Queues and Messaging Protocol between Linux processors

   In order to minimize the changes to other parts of the system software, our plan entails using the standard shared memory messaging queue's implemented today to communicate between TXRX, FP and SSC processors. The path of minimal disruption is to leave unchanged the message types and formats used today.

   This task addresses the changes required to the Linux kernel to initialize the shared memory queues and to add support to send and receive messages using standard Linux device drivers. In addition modifying the management-bus driver and EEE protocol modules are described below.

   It is a requirement that access by the SSC to the TuXRX 1480 mailbox registers is provided so the CPU can be immediately notified when a message is written into the shared queue. This task requires more investigation into the exact set of changes needed to enable this feature. It might be the case that the PROM code requires changes to do proper mapping of the PCI registers, etc.

4. Modify SSC "mgmt-bus" driver and EEE protocol modules to TuxRx Linux Kernel

Since we are replacing the TXRX's EEE functionality with a Linux kernel, this task covers the modifying of the SSC **mgmt-bus** device driver to the needs of the Linux kernel on the 1480, implementing the shared message queues between the SSC and 1480, and using common Linux device drivers and protocol modules on both the SSC and TuxRx Linux kernels.

The Linux **mgmt-bus** device driver and **eee** network protocol modules were written and integrated into the 4.0 release, hence this task is envisioned as a simple modification and testing effort.

5. Test the **mgmt-bus** driver and **eee** protocol modules on TuxRx Linux

   The task covers testing the **mgmt-bus** driver and **eee** protocol modules by running traffic between the SSC and 1480 after the various supporting software has been converted.

6. Memory Allocation Task

   This task covers the memory allocation interfaces, sizes and mapping functions currently used in the TXRX software. Since we are replacing the TXRX's EEE functionality with a Linux kernel on the 1480, the EEE memory allocation schemes must be explicitly addressed.

   Currently the EEE supports two memory regions, one for descriptors and buffers and the other for general memory allocation. The use of common shared memory regions mapped into all the cores must be maintained for descriptors, buffers, queues and messages. However other local memory allocations should be converted to call the generic Linux kernel memory allocator. The recommendation is to convert the eee_ramAlloc() and cache_alloc() interfaces into calls to the generic Linux kernel memory allocator.

   The plan is to allocate the skb's and their associated buffers from the common memory region so that the zero copy networking/filesystem operations are maintained. In addition, the allocation of the shared queue's and their associated messages must be allocated from another part of this common memory region to maintain backward compatibility.

7. Sibyte 1480 Processor Exception Handling

   This task covers the design and investigation of the existing MIPS Exception handling code on the 1480 with the replacement of the EEE OS by a stock Linux kernel.

   In the current software, the cores establish "passive crash" interrupt handlers which are activated by a mailbox interrupt when a processor core crashes. This approach supports the stopping of all cores in the event of a crash.

With the Linux kernel, a similar scheme will be required.

It is envisioned that the existing code will require modifications to use the mailbox interrupts for more tasks than just the "passive crash" case today. Some cases to consider include modifications to the FP code to interrupt the TuxRx whenever messages are placed in the IPC queues, or when the number of TXRX buffers/descriptors available for allocation on FP falls below some threshold. There are probably other cases which must be determined and properly handled.

8. Linux kernel debugger and core dump support

   This task covers obtaining a working kernel debugger kgdb, and other tools supporting obtaining MIPS kernel crash dumps on both the SSC and NCPU. The minimum requirement is to obtain a panic message which includes the processor registers, stack trace at the minimum for debugging. Having a debugger is a requirement for faster development.

9. Port RCON support to Linux Kernel

   This task covers adapting RCON SSC Linux driver to the TuxRx Linux kernel.

10. Test the RCON functions between SSC and TuxRx Linux

    The task covers testing the remote console (RCON) functions between the SSC and TuxRx after the various underlying supporting software tasks, covered previously have been ported and unit tested.

11. Distribution of messages from SSC.

    This task covers the messaging communication between the SSC, the TuxRx and the FP. Currently the TXRX receives all messages destined for the TXRX and FP coming from the SSC. The TXRX is responsible for forwarding messages destined for the FP.

    If we can remove the examination of the messages destined for the FP by the TXRX coming from the SSC, we can reduce the extra overhead in touching all of the FP messages. This task needs further study to accurately scope the implementation effort.

12. TuxRX Linux IP Forwarding Functionality

    This task covers the IP forwarding functions supporting sending packets to/from the SSC when packets are received on network interfaces supported by TuxRX Linux. It is envisioned that stock Linux kernel IP forwarding functions (netfilter). The netfilter functionality readily supports address translation, packet filtering and forwarding across interfaces typically used in firewalls, etc.

This task needs additional study to accurately scope the implementation effort.

13. Socket communication between TuxRx and FP

    This task covers the messaging communication between the TuxRx Linux kernel and the FP. The specific messages sent between the TuxRx and FP are defined in sm-tpl-fp/tpl-fp-api.h and cover socket operations such as open, close, listen, accept, read/write and unbind.

    The task requires supporting these messages in the TuxRx Linux implementation.

14. Virtual Stack communication between TuxRx and SSC

    This task covers the messaging communication between the SSC and TuxRx Linux kernel specific to virtual stacks. It covers the requesting and obtaining information pertaining to virtual interfaces, adding and deleting routes and obtaining routing tables, configuring interfaces, getting packet and network interfaces statistics, TCP and UDP connections, etc.

    Since the TuxRx will field these messages and generate the appropriate replies, this task addresses implementing the code to obtain the equivalent data from the Linux protocol stack. The messages and the current implementation are described in sm-ipm/ipm.[h,c].

    There are a number of messages that require a considerable amount of information to be passed back to the SSC regarding the state of the entire protocol stack. These include cumulative IP, UDP and TCP statistics, UDP and TCP connection tables with the message sizes ranging from 32KB to 800KB for statistics. Since some of these messages include information specific to the BSD protocol, this may require considerable work to maintain exact conformance to the existing message formats under Linux. Modifications of the messages in this area might be required.

    This task needs further study to accurately scope the implementation effort. One view is to implement portions of this functionality as a user level daemon which obtains the various statistics and connection information by reading either the /proc or /sys file system entries to get the information needed for the response message sent back to the SSC.

15. Virtual Server Support

    This task covers the messaging communication between the Virtual Server software running on the SSC and the TuxRx Linux kernel.

The Virtual Server message formats will remain unchanged, so the work covers implementing the functionality previously added to the BSD protocol stack on the TXRX which supported these messages.

The development centers on obtaining the information needed to satisfy requests and responding with appropriate replies. The Linux implementation must also implement those messages requiring explicit notification of changes in the networking stack occurring which must be communicated back to the SSC Virtual Server.

The Linux implementation of the vstack partitioning of the BSD protocol stack into separate instances may be implemented using Linux netfilter functionality. This is an open question needing more detailed study to scope the implementation effort.

16. Convert OnStor Packet Descriptors (pkt_desc) to Linux Socket Buffers (skb's)

    The task covers replacing the use of the pkt_desc data structure used in describing network data passed between the TXRX cores with the use of standard Linux socket buffers. This appears to be a straight-forward replacement, since they are both nearly equivalent and allows passing Linux networking buffers to the ACPU thread without copying.

    The task covers the kernel changes required to modify the skb memory allocator to use the common mapped shared memory region between the Sibyte 1480s versus using a generic kernel slab allocator region.

    A chain of skb's will be allocated from the common mapped shared memory region between the TuxRx and FP and continued use of zero-copy networking will be supported. The handoff of ownership of the buffers to the destination code via IPC using the shared messages queues, will continue.

17. Convert Linux kernel TCP/IP networking stack to be TPL-API aware

    This task covers modifying the Linux networking code to be aware of the Transport Layer API (tpl-api) interfaces that must be supported to communicate with the ACPU thread.

    This will allow the Linux networking code to call the appropriate tpl-api functions when changes occur requiring notification or reception of messages in either direction. It would be nice to avoid these level of changes to the Linux protocol stack if possible. Hence some additional investigation on other possible approaches is necessary.

18. Convert ACPU NFS/CIFS code to be a Linux kernel module

This task covers modifying the ACPU NFS and CIFS code to become standard Linux kernel modules. The additional changes envisioned include running as a Linux kernel process pinned to a processor core.

19. Convert ACPU NFS code to use Linux Socket Buffers

    This task covers modifying the NFS code to use Linux Socket Buffers (skb's) rather than OnStor pkt_desc's. Since the queue's and message formats will remain unchanged, with the exception of passing 'skb' pointers in the data messages, the basic assumption of passing a complete RPC/XDR message chain between the subsystems remains unchanged.

    A closer examination of the NFS code shows that is currently handles chains of buffers and uses only a few fields of the pkt_desc data structures which have equivalents in the skb data structure.

20. Convert ACPU CIFS code to use Linux Socket Buffers

    This task covers modifying the CIFS code to use Linux Socket Buffers (skb's) rather than OnStor pkt_desc's. Since the queues and message formats will remain unchanged with the exception of passing 'skb' pointers in the data messages, the basic assumption of passing a complete message chain between the subsystems also remains unchanged for CIFS.

21. Test modified ACPU NFS code

    This task covers testing the modified NFS code running under the Linux kernel on the ACPU processor core. Since the virtual server functionality is required by the NFS code, the testing and debugging of the modified code must be done later in the schedule.

22. Test modified ACPU CIFS code

    This task covers testing the modified CIFS code running under the Linux kernel on the ACPU processor core. Since the virtual server functionality is required by the CIFS code, the testing and debugging of the modified must be done later in the schedule.

23. NCPU Linux Performance Profiling Tools

    This task covers obtaining a working set of performance measurement tools for collecting data on he Sibyte 1480 processors. These would include the normal Linux Oprofile, GNU gprof and the Broadcom Sibyte profiler tools available from Broadcom under license.