I₂O :
A Platform For Driver-Level
Network Caching Investigation

by
Matt White

Computer Engineering Program
California Polytechnic State University
Dr. Jim Harris
# Table of Contents

Acknowledgments 1  
Abstract 2  
Introduction 3  
Project Description 5  
Introduction to I₂O 6  
  Description 6  
  Motivations 6  
  Primary Objectives 8  
  Hardware Architecture 10  
  Software Architecture 13  
Real World I₂O Hardware 16  
  PCI Bus 16  
  The i960 RP I/O Processor 17  
The IQ-SDK Development System 19  
  Hardware 19  
  Software 20  
Development Cycle Example 23  
Conclusions 27  
Bibliography 29  

Appendices

A. Source Code 31  
B. Disassembly Output 34  

List of Figures

I₂O IOP Configurations 12  
I₂O Device Driver Model 13
Acknowledgments

I thank Jesus Christ for literally saving my life, and for making it possible for me to attend school full time, work near full time, and raise a family of three. Without him, my participation in this project would have been impossible.

I thank my wonderful wife for her patience, understanding, and support during my entire educational adventure, and especially during this project. Without her support, this project would not have been a possibility for me, as I would have never made it this far. Her willingness to postpone her own education in order to take care of our two children for the past three years has been appreciated more than words can express.

I would like to thank 3Com for funding this project, and for their boldness in forming this unique relationship between the University and themselves. 3Com's involvement with the University, through this project, has already profoundly affected the educational experience of many students, including myself. Special thanks go to Bill Huber from 3Com for taking on the responsibility of directing the project on 3Com's side. His understanding of University schedules, and his willingness to work around them has been much appreciated. Special thanks also go to Jimmy Ruane of 3Com for his participation in the project.

Many thanks go to Dr. Jim Harris for his time and energy in directing this project and for dealing with all of the headaches that accompany project direction. I would also like to thank him for his dedication to the development of the Computer Engineering program, of which I am a direct beneficiary. This program provides students with an education in both hardware and software, which provides the perfect educational foundation for projects such as this. I would also like to thank Jim for his assistance in the completion of this document, and for his encouragement and advice on many occasions.
Abstract

This senior project is part of the first step towards the completion of a larger project funded by 3Com, under the administration of the Cal Poly Foundation, and directed by Dr. Jim Harris of the CPE program.

The long term focus of the project is to explore the application of the hierarchical memory model used in computer architecture to network performance. In order to carry out this research, knowledge of many existing technologies is required. Knowledge of the PCI (Peripheral Component Interconnect) bus, I2O (Intelligent Input/Output) architecture, the Intel i960 RP I/O processor, Windows NT device drivers, network performance benchmarks, network communications, embedded real-time systems, Wind River Systems' Tornado and IxWorks, and caching theory are all required for the success of the project. This is obviously beyond the scope of any one senior project, and thus the research will span many senior projects, with each new group of students concentrating on a particular aspect of the project.

The primary project goals during this senior project were the establishment of a network test bed, and the installation and configuration of an I2O capable development system. The IQ-SDK development platform from Intel, which contains an Intel i960 RP I/O processor board and software cross-development tools was installed as part of this process. Study of the I2O architecture, PCI bus, and the Intel i960 RP I/O processor has been a large part of this senior project, as they are the underlying technologies associated with the development system, and the research efforts in general.

This document will provide a high level introduction to the I2O architecture, and consider the use of the PCI bus and the Intel i960 processor as used to implement an I2O subsystem. It will also provide a description of the content and capabilities of the IQ-SDK evaluation platform, and an example of the development process using these tools.
Introduction

The introduction of the World Wide Web (WWW) has brought the Internet into the main stream of society. Gone are the days when only a small number of people from Government agencies and Universities traveled the "Super-Highway". Now, nearly every company from small to large has a World Wide Web site, and more and more business is being conducted over the WWW. This explosion in Internet usage has created a huge demand for technologies providing increased data rates, increased throughput, and most importantly, increased end-user response time. Technologies such as ATM (Asynchronous Transfer Mode), Fast Ethernet, and Gigabit Ethernet have all been created in response to this demand.

Most of the technology improvements have been in data rate and throughput, but these alone can only increase the end-user response time to a point. Even if the network data rate were to increase to infinity, end-user response time would not go to zero, because other factors exist that must be taken into account. Factors such as propagation delay, I/O subsystem bandwidth, and host processor loading all affect the end-user response time. As the data rate increases, its contribution to the overall end-user response time becomes smaller and smaller, thus making it more important to focus on improving the other factors: I/O subsystem bandwidth, and host processor loading.

Currently both I/O subsystem bandwidth and host processor loading, especially frequent I/O interrupts, are limiting factors in end-user response time improvement. Increasing the system bus bandwidth, and making processors more powerful would be a brute force answer to the problem, but a more elegant and less technology dependent solution may be found in smarter I/O subsystems capable of offloading the host CPU from numerous I/O interrupts, and providing an ability to cache network data. By intelligently dealing with input before shipping out to the host processor, the amount of bus bandwidth used can be reduced. By caching network data, redundant network accesses can be avoided, thus decreasing the number of total network transactions. Without any increase in data rates, bus bandwidth, or processor power, the end-user should see a shortened response time, due to a decrease in both the number of network transactions, and in the volume of data being transferred to and from the network and across the
Intelligent NICs (Network Interface Controllers) are already emerging, but are plagued by a lack of standardization and dependence upon specific system topology and bus architecture.

I²O (Intelligent Input Output) is a somewhat recently introduced I/O architecture which may prove to be an excellent environment for the exploration of network caching and intelligent I/O subsystems.
Project Description

The overall project investigates the effects of applying the hierarchical memory model used in computer architecture to network communications - more specifically, how network caching will affect the response time to user requests for data. This goal requires many initial tasks. The most important of these being:

a. the establishment of a network test bed,

b. baseline network performance measurements,

c. port a 3Com Windows NT device driver to the Intel i960 RP based I2O subsystem,

d. comparison tests with new device driver.

Currently (a) is nearly complete. Only a few more software and hardware items remain to be acquired before (a) is complete. Part of (a)'s completion was the installation of Intel's IQ-SDK evaluation platform, which was completed as part of this senior project.

The next step after installation of the IQ-SDK evaluation platform was to develop a demonstration program to become acquainted with the development tools, and to verify the proper operation of the evaluation board. A program was written to communicate to the host computer via the monitor interface, and light LEDs on the evaluation platform as requested by the user. This program is listed in appendix A of this document, and is discussed in the Development Cycle Example section.

In addition to the steps listed above, it was determined that a document providing an introduction to I2O and the IQ-SDK development tools would be useful. The following chapters will provide this introduction and provide a list of resources for more in-depth coverage of the topics.
Introduction to I₂O

I₂O is a specification for an intelligent, standards based, OS and device independent input/output subsystem architecture [I₂OSIG3]. The guiding philosophy of the specification is to provide a more intelligent I/O subsystem, that is capable of relieving the host processor from much of the I/O burden.

Motivations

Three problems associated with current I/O architectures are the primary influence for the development of the I₂O specification. These are: device driver proliferation, I/O bottleneck, and the lack of I/O architecture standardization.

Device Driver Proliferation

The device driver proliferation problem refers to the current dilemma faced by device vendors when introducing a new device. A device driver must be written for every version of every operating system that the device intends to be used under. If the device vendor wishes to provide support for six operating systems, and each of these operating systems has 6 versions, then the device vendor must develop 36 device drivers (This is by no means an over-statement of the problem. Consider the plethora of operating systems currently in use: MS-Windows 3.1, MS-Windows 95, MS-Windows NT, MS-DOS, many versions of UNIX, IBM's OS/2, Novell's Netware etc. Each of these operating systems has many versions which are incompatible at the device driver level), Needless to say, this is a waste of resources; resources that could be better spent developing more innovative devices.

I/O Bottleneck

As network adapters, disk controllers and other I/O devices increase in speed, and as
the number of devices per system increases, system I/O throughput becomes a bottleneck in I/O performance. The 32-bit wide, 33 Mhz PCI (Peripheral Component Interconnect) bus had provided a good solution to the I/O subsystem bandwidth problem in the recent past, but ever-increasing NIC (Network Interface Controller) data rates, faster storage device bus rates (i.e. Ultra SCSI, EIDE), and higher performance video adapters are all eating away at the PCI bus bandwidth. The 64-bit, 66MHz implementation of the PCI bus will help alleviate the I/O bus bottleneck by providing a four-fold increase in bandwidth, but in the near future, even this increased bandwidth of 4.192 Gbps (132Mbytes/sec * 4 * 8 bits/byte) will not have sufficient capacity for servers of Gigabit Ethernets which could require bus bandwidth of up to 8 Gbps for a four subnet server (four Gigabit Ethernet cards @ 1 Gbps input and 1 Gbps output * 4). Adding to the problem of limited bandwidth, is the host processor's inability to deal with more than one interrupt at a time. In cases where the I/O comes in many small bursts, each causing a host CPU interrupt, it is possible for the system bus to be unsaturated and still have an I/O bottleneck due to the host processor's inability to service the I/O interrupts efficiently. This situation also causes the host CPU utilization to increase dramatically, which will affect the end-user response time in an adverse manner.

Lack of I/O Architecture Standards

Currently, there are a plethora of standards that cover individual pieces of the I/O subsystem. The PCI standard covers the system bus, operating system vendors such as Microsoft require certification of device drivers, and numerous I/O device standards exist, but no one standard has covered the entire I/O subsystem architecture in a non-proprietary manner. This leads to a tremendous incompatibility problem, and is the cause of the device driver proliferation problem mentioned previously. Without a standard I/O subsystem architecture, each device driver must be re-written for every flavor of operating system, and for every new device. The huge base of existing devices and device drivers that work under current, non-standard I/O architectures, make switching to a new I/O paradigm seem impossible. Will all of these devices and device drivers need to be thrown away if we change the I/O architecture in such a drastic manner? No, not if the change is made such that legacy devices and device drivers may operate under the new paradigm without modification.

The push for intelligent I/O devices which use technologies such as DMA and peer-
to-peer communication to bypass the host CPU adds to the looming I/O architecture incompatibility problem. These technologies are powerful in their ability to relieve the host processor from I/O processing, but have the dangerous potential of indiscriminantly eating up large amounts of system bus bandwidth if not implemented properly. How will delegation of bus bandwidth to these fighting intelligent devices be handled? This kind of I/O device intelligence cries out for a standard architecture to provide a framework for the organization and delegation of system I/O resources. In addition, such a standard could provide a common interface to I/O devices and provide a centralized intelligence for I/O devices so that each new device would not need to be autonomously intelligent.

**I₂O Objectives**

The I₂O specification [I₂OSIG3], generally speaking, aims to provide a standards based I/O subsystem architecture that is independent of both the host operating system and the particular device being controlled. It attempts to provide an answer to the problems plaguing I/O subsystems of the day, as mentioned in the previous section.

*Operating System and Device Independence*

The I₂O specification is designed to be entirely independent of the host operating system, at least from the perspective of the I/O subsystem. By doing so, it provides a common interface between the host and the I/O subsystem which enables source level device driver portability across operating systems. Only the small portion of the device driver that must communicate to the host OS needs to be re-written when porting to another OS. The portion of the driver that actually speaks to the I/O sub-system is completely standardized, and will not change.

The specification is also written to be independent of any specific I/O device. It does so by logically splitting the device driver into two portions. One portion runs under the host operating system and contains a small amount of code that is specific to the particular operating system, while the rest is dedicated to interfacing to the I/O subsystem. The other part of the
device driver is responsible for communicating to the specific device. It runs on the I/O sub-
ystem's processor which is running a real-time operating system that provides the services for 
communicating to the host side of the device driver. These two portions of the device driver are 
called the Operating System Specific Module (OSM) for the host side, and the Hardware Device 
Module (HDM) for the device side. This model allows device vendors to provide only one HDM 
when releasing a new device. This one HDM will work in any I2O compliant system, regardless 
of the specific host operating system or system topology.

\textit{I2O Subsystem Isolation}

Provision for isolation of the I/O subsystem from the environment of the host 
operating system is another objective of the I2O specification. This isolation is accomplished by 
providing an execution environment external of the host operating system, called an I/O platform 
(IOP). The I/O subsystem is then made up of any number of IOPs which are not directly under 
the control of the host operating system, but must be accessed through the I2O protocol layers.

\textit{Scalability}

Scalability and compatibility across different system topologies is also provided for in 
the specification. System topology cross-compatibility is accomplished by providing a layer of 
abstraction between the low level details of the system topology such as the system bus, the host 
processor type, and host processor configuration (single, parallel etc.) and the device driver. If 
the system topology changes, then only that portion of the I2O system code must be altered, while 
existing I2O device drivers will remain unchanged. Intelligent I/O subsystems will have a greater 
ability to scale with host processor power than their non-intelligent counterparts due to the 
hierarchizing of the I/O subsystem structure. Current I/O sub-system architectures do not scale 
well as higher performance host processors are used. This is caused by the I/O bottleneck, which, 
to a large degree, nullifies the effects of increased processor power on overall system 
performance.

\textit{Legacy Device Driver Compatibility}

The last of the primary objectives of the specification is to maintain compatibility with
the existing device driver model. Since the specification does not require singular control of the I/O sub-system resources, this is easily accomplished. Legacy device drivers will work in the same manner that they always have, right along side I\(_2\)O.

In summary, the main goals of the specification are:

- To specify an architecture that is operating system vendor independent and is adaptable to existing operating systems.
- To provide a I/O sub-system architecture that is independent of the specific I/O device.
- Enable source level portability across target processors for device drivers.
- Provide isolation of the intelligent I/O subsystem from the environment of the host operating system.
- Provide scalability across system platforms, and increased processor power.
- Maintain compatibility with legacy device drivers.

**Hardware Architecture**

The I\(_2\)O specification is independent of the particular host processor, system bus, I/O processor, and overall system topology. Even so, the system bus and I/O processor must both be able to perform certain tasks in order to be fully I\(_2\)O compliant.

The system bus must allow peer-to-peer communication, and support DMA (Direct Memory Access) in order for I\(_2\)O to be implemented properly. Although not a requirement, the system topology should be such that peer-to-peer transactions can be carried out in parallel with host CPU main memory accesses. This will ensure optimum performance, as the intelligent I/O subsystem will be able to accomplish much work without interrupting or interfering with host processor execution.
The I/O processor must have a readily available C compiler, in order to support loadable device driver modules. This is the only requirement that the specification imposes on an I/O processor. Although not explicitly stated, the I/O processor must also have the capability of running the I/O embedded RTOS (Real-Time Operating System). This implies certain capabilities such as software interrupts, and sufficient address space to accommodate the RTOS. Other capabilities that make I/O processor candidates more attractive for simple implementation of an I/O subsystem are: sufficient processing power for potentially intense I/O activity, fast interrupt response time, on-board DMA controller, and debugging capabilities.

I/O Platforms

Intelligence in an I/O system lies in two types of nodes: host nodes, and I/O processor nodes. A host node, as defined in [I2OSIG3] is "... one or more application processors and their resources executing a single homogeneous operating system." This definition obviously leaves room for a number of host configurations including single or multi-processor configurations, or even a distributed operating structure, as long as the processor(s) are executing a single operating system. An I/O processor node consists of a processor, memory, and I/O devices, and is termed an I/O platform (IOP). IOPs are dedicated to processing I/O transactions and can be physically arranged in a number of ways, however, I/O is optimized for a single host node and a number of IOPs.

Even the structure of the IOPs is left wide open by the specification. Figure 1, on the following page, from [Intel2], illustrates the two most common IOP configurations. An IOP may or may not have its own expansion bus. It may control devices on the system bus, or on its own expansion bus, or both. With this vagueness in specification of structure, it is important to realize that regardless of the physical configuration of an IOP, it is still one logical unit in the overall I/O subsystem.

In figure 1, IOP 1 is setup such that it's devices are isolated from the system bus and only accessible through the I/O processor. IOP 2 contains devices that sit on the system bus, and are therefore visible to the entire system. Devices A and B represent devices under the old model which are accessed directly by the host processor.

IOPs could be implemented as follows:
• The I/O processor, and it's memory, resides on the motherboard and is designed to control devices on the system bus, embedded devices, and devices on its own expansion bus, thus making the entire I/O subsystem one large IOP.

• The I/O processor, and it's memory, resides on an add-in card which sits on the system bus. The devices under it's control reside on a local expansion bus that is isolated from the system bus. In this case the IOP is encapsulated on the add-in card, and any access to it's I/O devices must be through the I/O processor.

• The I/O processor resides on an add-in card dedicated to a specific purpose (such as a network interface adapter). The I/O processor's singular task is to control the functions of the adapter that it resides on.

The first case listed is unique in that some devices that lie in the physical realm of the I/O processor may not be under it's control. These devices, residing on the system bus, would be under the control of the host CPU, using the old I/O device driver model in which the entire device driver executes on the host CPU. The other two cases isolate their I/O devices from the system bus, thus making it impossible for the host to access them without going through the I/O processor. A fourth configuration could exist in which the I/O processor resides on an add-in card, and controls I/O devices on both it's own local expansion bus, and devices on the system bus.
Software Architecture

At the core of the I₂O software architecture is the split device driver model. The existing device driver model consists of one monolithic device driver executing on the host processor under the host operating system. The split device driver model splits the functionality of the device driver into two pieces: the Operating System Specific Module (OSM), and the Hardware Device Module (HDM). Figure 2 is a graphical comparison of the current monolithic model and the I₂O model taken from [Intel2].
Today

![Diagram of I2O Device Driver Model](image)

**Figure 2: I2O Device Driver Model**

*Operating System Specific Module*

The OSM interfaces to the host operating system and is the link to the I/O subsystem, although it contains no device specific code. As far as the host operating system is concerned, the OSM is just another device driver. The I2O specification groups devices into classes so that only one version of an OSM for multiple HDMs of that I/O class is necessary. In addition, the OSM does not need to be configured for a specific vendor's HDM. As an example, consider the network interface class. For all of the network interface cards in the system, the same OSM would be used to provide access to the particular device driver modules (discussed below) for the devices. Each DDM and network card could be from a different vendor, say 3Com, Intel, and Racal. Since each of these devices provides the same kind of services to the host, the interface from the OSM to the HDM can be identical. The I2O specification includes several classes such as: random block storage devices, sequential storage devices, SCSI peripherals, SCSI adaptors, LAN adaptors, Fibre Channel peripheral, Fibre Channel port, etc.
Hardware Device Module

The HDM contains all of the device specific code. It handles all of the low-level configuration and direct communication with the device. This split in the device driver model, allows for source level portability of the device driver across processors. Only a small portion of the OSM must be re-written for the operating system specific interfacing. The HDM remains the same regardless of the host specific operating system. In addition to these two basic types of device driver modules, the specification mentions two others. The Intermediate Service Module (ISM) provides some functionality between the OSM and the HDM. Many of these can exist between the OSM and HDM to provide a stackable device driver structure. The utility of this intermediate type of module is not clearly described in the specification. The other type of module used in the specification is the Device Driver Module (DDM), which is simply the combination of any ISMs and the HDM for a particular device. Throughout the specification, DDM is the preferred term when referring to the device side portion of the device driver.

Message Layer

The OSM and HDM communicate with one another via the I2O message layer. The message layer also provides peer-to-peer communication between DDMs, whether they reside on the same IOP or separate IOPs. The message layer provides the following services:

- Message delivery between I2O modules anywhere within the system
- Configuration registry
- Abstraction from the physical platform

The messaging layer is made up of a network of MessengerInstances, which provide for initialization, configuration, and operation of its client modules. The MessengerInstance is not a single process, but a collection of services. Each I2O node, including both the host node(s) and the IOP(s), has one set of MessengerInstance services running.
The messaging service provided by the MessengerInstances are accessed by application program interfaces (API) provided by the I/O real-time operating system (IRTOs). In addition to messenger service APIs, the IRTOS supplies other services to the DDMs such as:

- Memory allocation
- DDM and device registration
- DMA operation
- Bus access
- Timer operations
- Interprocess communication facilities
- Semaphores

The IRTOS allows multiple DDMs to run on a single IOP, providing efficient task switching, and most importantly, real-time response to I/O interrupts. It is also the manager of IOP resources. The IRTOS and the APIs that it provides for interfacing with other I2O nodes, makes it possible for vendor supplied HDMs to be independent of the specific host operating system. As long as the provided APIs are used for communication and system resource allocation, then the device driver will be source code portable across platforms, regardless of the specific I/O processor or the particular host operating system. Currently, only Wind River Systems' IxWorks IRTOS is compliant with the I2O specification [I2OSIG3].
Real World I\textsubscript{2}O Hardware

As stated before, the I\textsubscript{2}O specification does not require any particular system bus, I/O processor or particular system topology. It is true, however, that certain characteristics in a system bus, and in the I/O processor, make them more attractive for use in the implementation of an I\textsubscript{2}O system.

Peripheral Component Interconnect Bus

Of the system busses that have recently been in wide use, the peripheral component interconnect (PCI) bus is most suited for the implementation of I\textsubscript{2}O subsystems [Shanley]. Several characteristics of the PCI bus make it the bus of choice for I\textsubscript{2}O implementation. These characteristics are:

- DMA capabilities
- Bus mastering capabilities
- Peer-to-Peer communication
- Resource sharing
- High bandwidth (132 Mbytes/sec burst)

The DMA capabilities of the PCI bus allow IOPs to place data directly into main memory without the assistance of the host processor, allowing the host processor to continue accessing its level two cache in parallel.

Peer-to-peer and bus mastering capabilities enable intelligent devices and IOPs to control the bus and communicate to one another. This could be used by intelligent network adaptors to route packets to other adaptors without the assistance of the host processor.

The I\textsubscript{2}O messaging scheme relies heavily on shared memory, and as such, the shared
resource capabilities of the PCI bus are necessary for I2O conformance. Another capability of the shared resource mechanisms provided by the PCI bus is the ability to lock a resource. This can be used when multiple intelligent devices need to access a shared resource, such as a mass storage device or main memory. This locking mechanism is identically analogous to the semaphore locking provided by many operating systems.

*Intel i960 RP I/O Processor*

Currently, only the Intel i960 RP processor contains all of the features necessary for full I2O compliance on one chip [I2OSIG1]. This does not mean that the i960 RP is the only processor available that can be used as an I2O I/O processor. Other processors can be used, but will require other integrated circuits to support the functions necessary for the implementation of an IOP. The convenient, and powerful thing about the i960 RP is that it contains all of the necessary functionality, including many extra bells and whistles, in a single processor package.

The i960 RP includes the following features on board:

- PCI-to-PCI Bridge
- Messaging Unit including 4 message registers, 2 doorbell registers, 4 circular message queues, and 1004 index registers
- Two Address Translation Units (ATU) which connect the local bus to the PCI buses
- Memory Controller supporting up to 256 Mbytes of DRAM in many configurations
- DMA Controller with 2 channels dedicated to the primary PCI bus, and 1 channel dedicated to the secondary PCI bus
- I/O Advanced Programmable Interrupt Controller (APIC) Bus Interface Unit providing multiprocessor interrupt management for Intel CPUs
- Inter-Integrated Circuit (I²C Serial) Bus Interface Unit
• Secondary PCI Arbitration Unit supporting up to six secondary PCI devices

In addition to these functional units, at the center of the i960 RP is an 80960JF processor core. This processor core is capable of executing one instruction per clock and has a 4 Kbyte two-way set associative instruction cache, a 2 Kbyte direct-mapped data cache, 16 global registers, 16 local registers, two 32-bit timer units, and a local register cache with 8 available stack frames [Intel4]. The i960 RP has more than one hundred memory mapped configuration registers for setting the numerous functional units and core processor functions. The i960 RP also contains 1Kbyte of internal data RAM on chip.

The PCI-to-PCI bridge unit, address translation units (ATUs), messaging unit, and DMA controller provide the functionality necessary to set up an IOP in nearly any conceivable fashion. The PCI-to-PCI bridge can be configured to isolate the secondary PCI bus from the first, making its devices visible only to the i960 and the other devices on the secondary bus. It can also be configured to logically join the system PCI bus to the secondary PCI bus, thus extending the original system bus. In this configuration, all PCI bus messages whose target address resides within the configured window of addresses are forwarded. The ATUs provide conversion of i960 local bus addresses to PCI bus addresses and vice versa. The DMA controller provides the i960 core with DMA access on both the primary and secondary PCI buses.

Another important feature of the i960 RP is its interrupt response time. This is of primary importance for an I/O processor since servicing I/O interrupts is its most important task. The 33 Mhz i960 RP processor is capable of interrupt response times less than one microsecond if the provided interrupt latency improvement schemes are used [Intel4]. These schemes include caching interrupt vectors on chip, caching portions of interrupt handling code, reserving register frames in the local register cache, and caching the interrupt stack in the data cache. The only potential drawback to using these schemes is that all of them take up processor cache resources that might improve overall performance of foreground execution. Depending on the nature of the application, using these schemes may or may not be advisable, although for most I/O tasks, the decreased interrupt latency is more important than the minor increase in foreground execution performance.
The IQ-SDK Development System

The IQ80960RP Evaluation Platform, also known as the IQ-SDK (Software Development Kit), is a development kit distributed by Intel, intended to be used to develop intelligent I/O subsystems (but not necessarily I2O compliant subsystems). The kit contains both a hardware platform and software cross-development tools. However, the kit does not contain everything necessary to build a fully I2O compliant I/O subsystem. The missing link is Wind River Systems IxWorks IRTOS, and the accompanying integrated development environment, Tornado for I2O. A license for IxWorks and a 30 day evaluation copy of Tornado is supposed to be included in the purchase of every evaluation platform, but due to newness of the Wind River software and the method by which the University acquired the evaluation package (donated by Intel, not purchased), these were not included with the evaluation platform provided to the University.

IQ-SDK Hardware

Two separate expansion boards were included in the IQ-SDK package. One is a full length PCI expansion board that contains all of the hardware necessary for a flexible, fully I2O compatible IOP. The board, although distributed by Intel, was developed and is manufactured by Cyclone Microsystems. It hosts an i960 RP processor, two standard 72 pin SIMM slots populated with 2 Mbytes default but expandable to 64 Mbytes, a secondary PCI bus slot, one RS-232 serial port, one JTAG connector, one I²C serial interface connector, one APIC connector, two flash memory sockets, one 512Kbyte serial EEPROM attached to the I²C bus, an emulator header, and eight user LEDs [Intel8].

One of the flash memory sockets is populated with a ROM containing the MON960 monitor. The remaining socket contains a flash ROM available for user applications. Switches are provided to change the socket from which the processor will boot. Default is to boot the processor using the MON960 ROM, but the processor can also be booted from the user flash ROM or from a device connected to the emulator header.
The secondary PCI bus slot differs slightly from a standard PCI bus slot in its physical layout as well as the pin assignments. Four clock signals, four request signals, and four grant signals have been added to the connector replacing six signals from the standard PCI edge connector plus six reserved pins [Intel8].

The RS-232 serial port is based on a 16C550 UART and is capable of transfer rates from 300 baud to 115,200 baud. It is particularly useful in that an external system can be connected to the I2O platform to perform diagnostics, while the IOP's host system is allowed to operate normally without the unnatural load of running IOP diagnostics.

The other expansion board included in the kit is a combination SCSI/Ethernet card designed to fit into the secondary PCI bus expansion slot on the main board. The board provides two 16-bit Ultra SCSI (40 Mbps) ports, and two 10/100BaseTx Ethernet ports. The SCSI ports are capable of standard 8-bit SCSI transactions, wide-SCSI, and Ultra SCSI. The Ethernet ports are auto-detecting 10 Mbps or 100 Mbps (Fast Ethernet) ports [Intel9].

Additional peripherals are available for the evaluation board from the manufacturer of the board, Cyclone Microsystems [Cyclone].

**IQ-SDK Software Development Tools**

IQ-SDK comes with some basic software cross-development tools as well as some example programs. The included tools, listed in [Intel5], are:

- CTOOLS/960 C Cross Compiler
- MON960 Monitor
- GDB960 Debugger

The CTOOLS/960 cross compiler is a fully ANSI compatible optimizing C compiler that is compatible with the GNU/960 C compiler [Intel1]. The compiler uses a plethora of target processor independent optimization schemes including: constant expression evaluation, constant
propagation, collapsing of arithmetic and bitwise boolean identities, common subexpression elimination, register subsumption, local variable promotions, tail-call elimination, procedure inlining, branch optimizations, dead code elimination, loop invariant code motion, variable shadowing, superblock formation, and basic block rearrangement [Intel5]. In addition to the processor independent optimizations, the compiler performs many target processor-specific optimizations such as specialized instruction selection, code scheduling, use of the i960RP's on chip data RAM for spill registers, and efficient use of complex addressing modes, among others. Whole program optimizations, such as function inlining across source files, and profile driven optimizations are also supported by the compiler.

The CTOOLS/960 package also includes an assembler and linker, which combined, take the output of the compiler and produce an executable file. The CTOOLS package includes many other useful utilities including:

- An archiver for building libraries
- A disassembler
- A ROM builder for producing code suitable for burning into a ROM
- A statistical profiler for determining the percentage of execution time that each portion of the code is taking
- An object file stripper for removing debug information from object files
- COFF to IEEE-695 object file converter

Another major part of the development tools provided with the IQ-SDK is MON960, an on-board monitor program that provides the execution environment on the i960 evaluation board. On boot up, MON960 initializes the memory controller, the primary and secondary ATUs, the PCI-to-PCI bridge unit, and the i960 processor core. MON960 also initializes the secondary PCI bus and any devices located on it. This involves allocating memory mapped I/O address space for each device, assigning PCI base addresses, and assigning interrupt request channels [Intel5].
MON960 provides two methods for down-loading application code to the i960 platform: through the RS-232 serial port or the PCI bus. It also provides single step execution, breakpoints, memory and register display, and disassembly of i960 code [Intel5]. The monitor program, as an execution environment, is not useful for final I2O implementations, but is useful during the proof of concept and test/debug stages in the development process. Even when the final execution environment is intended to be the I2O IRTOS, MON960 may be useful for debugging portions of code that can be isolated out of the IRTOS execution environment, thus eliminating any effects of the IRTOS.

The final tool included in the IQ-SDK package is GDB960, a full symbolic debugger, as well as a graphical user interface (GUI) for it. The debugger communicates with MON960 using the Host Debugger Interface Library (HDIL). The physical communication path between GDB960 and MON960 can be the RS-232 serial port or the PCI bus. In conjunction with MON960, GDB960 allows source level step execution, breakpoints, monitoring of variables, and monitoring of general purpose registers and status registers [Intel2]. The debugger also acts as the console for MON960 in that all standard input and output is through the debugger. If the i960 application code contains a printf, then the output will be seen in the debugger window. If the connection to MON960 is through the PCI bus, the monitor transmits the printf data to the host via the i960's message registers.

The GUI provides a convenient interface to the capabilities of the debugger. Some things, such as simultaneous viewing of registers, variables and source code while debugging, that are not possible with the text output of the debugger become possible with the GUI interface. Integrated into the GUI is a text editor with C syntax highlighting. The GUI also provides the ability to add menu items that execute external programs [Intel12]. These two features of the debugger GUI nearly turn it into an IDE (Integrated Development Environment). The only thing required is for the user to set up a batch file or makefile to build and/or compile the current project and add it to the extensible menu.
Development Cycle Example

This section will provide an example of the use of the IQ-SDK platform to develop, test and debug a user application for the i960 board.

The steps required in the cross-development process are:

1. Connect the debugger (on host) to the monitor (on target)
2. Create source code
3. Cross-compile, assemble and link to executable
4. Transfer the executable to the target
5. Run, test and debug the program
6. Make modifications as required
7. Repeat steps 3 through 6 as needed

The GDB960 debugger can be started by clicking on the Tornado icon. Don't let the icon fool you, this is not Wind River System's complete Tornado package. GDB960 is part of Tornado, but the entire package contains much more that is not included in the IQ-SDK package. Once the debugger has loaded, the debugger must be connected to the target. This connection can be made through the RS-232 serial port on the IQ80960 board or through the PCI bus. If the PCI bus is to be used, the PCI driver (PCI_WNT.SYS) must be installed on the system. Press the Target Connect button in the GDB960 buttons window. Next, select the type of connection to be made (PCI or serial). If PCI is chosen, then a window requesting the target device will come up. If you choose serial, then a window requesting the serial port parameters comes up. After completing these steps, the debugger will attempt to connect to the target. If the connection is successful, a window displaying the name of the target, and other information about the target will appear [Intel12].
The source code can be created with any text editor, however, the GUI for GDB960 comes with an integrated text editor featuring C syntax highlighting. Using the integrated text editor is more convenient than using an external editor, and allows for simultaneous viewing of the source code, execution output, and debug information.

Once the source code has been created, it must be compiled and linked. Using the CTOOLS960 C compiler provided with the IQ-SDK, compile the code. This can be done in a separate DOS prompt, but it is more convenient to set up a batch file to compile and link the current project, which can then be added to the extensible menu and executed from within the debugger environment. An alternative to a batch file would be to set up a make file. This is highly recommended for projects requiring more than just a few source files. GNU make, although not included in the IQ-SDK, is freely available and has been installed on the development NT machine. The command line for the compile command will be as follows:

```
gcc960 -ARP -Fcoff -c -g -o <outputfile> <sourcefile>
```

Here `<sourcefile>` refers to the name of the source file being compiled, and `<outputfile>` refers to the name of the object file which the compiler will create. The `-A` flag is used to tell the compiler the specific CPU architecture being used, so that it can perform architecture specific optimizations. In this case `RP` stands for the i960RP. The `-F` flag tells the compiler what file format to output. MON960 requires the COFF format, therefore we flag the compiler to generate a COFF object file. The `-c` option tells the compiler to compile the source file, but not link. This flag can be skipped if there is only one source file, in which case the compiler will compile and link the source file, producing the final executable file. The `-g` option causes debugging information to be included in the output file which GDB uses to provide source level debugging. The `-o` option tells the compiler that the next argument will be the name of the output file [Intel5].

If the `-c` option is used, then the linker must be run as well. The link command will be as follows:

```
gld960 -ARP -Fcoff -Tcyrx -o <outputfile> <inputfiles>
```

Here `<inputfiles>` is a list of object files to linked together, and
<outputfile> denotes the name of the executable outputfile. The first two flags, \texttt{-ARP} and \texttt{-Fcoff}, as well as the last flag, \texttt{-o}, serve the same purpose as they do for the compile command. The \texttt{-T} flag tells the linker to link for a specific target. In this case \texttt{cyrx} tells the linker to link for the Cyclone i960RP board.

Once the source file has been compiled and linked successfully, the executable file can be transferred to the i960RP evaluation board. This can be done by choosing the Open Binary item from the file menu. A dialog box with two check boxes for \textit{Load symbols} and \textit{Download code} will come up. Choose both the \textit{Load symbols} option and the \textit{Download code} option. The \textit{Load symbols} option will allow for source level debugging. An alternative method for invoking the download command is to press the icon showing an arrow pointing at a target. This will cause a file dialog box to appear. After choosing the desired file, press the download button in the dialog box. The file menu method is preferred, as it allows for loading of debugging symbols.

Now the target is ready to execute the code just transferred to it. The program can be executed under debugger control, or without the interference of the debugger. To run without debugger interference, simply press the run button or choose the run item from the Debug menu without setting up any breakpoints. To run the program under the control of the debugger, first set a breakpoint within the code, so that normal execution will stop at that point. To do this, the source code must be visible. Pressing the List Source button from the GDB960 buttons window will allow loading of any source module that is part of the current executable. After loading the source code, move the cursor to the desired breakpoint location. Pressing the F9 key or the icon with the solid arrowhead pointing down to a line will toggle a breakpoint at the current location. Single step execution can be accomplished by pressing the solid down arrow. A function can be stepped over by pressing the broken down arrow icon. To return to normal execution, press the double down arrow icon.

GDB960 also allows viewing of variables and expressions. This can be done by first highlighting an expression or variable name within the code, then selecting Inspect from the Debug menu. The inspect function of the debugger supports all of the C data types, including structures and unions. Choosing the registers item under the Debug menu will cause a window with all of the machine registers to appear. The registers and data structures displayed are displayed in a tree like fashion. Clicking on the name of a structure will cause it's contents to be
displayed. Clicking again on it's name will cause it's contents to disappear. Memory can also be viewed by choosing the memory item from the debug menu.

The debugger also allows viewing of the disassembled code. Choosing the Disassembly item from the View menu will produce a symbolic disassembly listing of the code. Choosing Mixed Source and Disassembly will display the disassembled output following each line of source code. Attempting to directly print this output fails. In order to print the disassembled code, the disassemble command must be run from the GDB960 command line interface, and the results copied out of that window into a new file window. The disassembled code can now be printed from this window.

Appendix A contains the source listing for a sample program that was used to test the development system and explore its abilities. This program flashes the LEDs on the evaluation board and then allows the user to choose an LED to light. The monitor/debugger interface is used for the standard input and output. This program can be used in the future to verify that the IQ80960 platform to host interface is working correctly.

Appendix B contains the disassembly listing for the source program listed in Appendix A. Close examination of this disassembly listing reveals that the quality of the C compiler is quite high. The compiler's output appears to be compact and efficient.
Conclusions

The I₂O architecture provides a framework for the exploration of intelligent I/O subsystems, and as such provides an environment for the exploration of the application of the hierarchical memory model used in computer architecture and caching theory to network performance, specifically end user response time. I₂O's support for peer-to-peer communications, direct memory access, and the common interface that it provides to these functions will be extremely useful in the implementation of network caching systems. I₂O's ability to reduce host CPU utilization by offloading I/O tasks to the I/O processor is also important due to the potentially CPU intensive nature of the network caching algorithms.

Cyclone Microsystem's IQ80960 evaluation board provides a fully I₂O compliant hardware platform. With the assistance of a PCI bus expansion card, the 3Com Boomerang NIC can be installed in the secondary PCI bus of the IQ80960, isolated from the rest of the system. This will make the Boomerang accessible only through the i960RP processor in control of the IQ80960 evaluation board, thus turning the Boomerang into an I₂O device. At this point, the exploration of network caching can be easily accomplished, with the assurance that the host CPU utilization will not increase, regardless of the complexity of the processing.

Wind River System’s Tornado development system and IxWorks IRTOS will provide the necessary software for full I₂O compliance on the IQ80960 evaluation board. The development tools provided in Tornado should prove invaluable in the development of the network caching device drivers.

Study of the I₂O architecture and associated technologies such as the i960 RP I/O processor, and the PCI bus have provided part of the technical foundation for stepping into the implementation phase of this project. Other team members are concentrating on Windows NT device driver technology, network performance testing technology, and network caching theory. All of this knowledge is essential for the successful completion of the project.

The installation and evaluation of the IQ80960 board and the IQ-SDK development
tools were an important step in the establishment of the network test-bed. The only remaining items for the network test-bed to be complete are: Wind River System’s Tornado, Cyclone Microsystem's three PCI slot expansion module for the IQ80960 evaluation board, the Future Plus PCI bus extender, Microsoft's Device Development Kit, and NuMega's Soft ICE. As of this writing, all of these items are being aggressively pursued.
Bibliography

[Brown]
M. Brown, "I2O Architecture Overview", Intel Corp.,
http://developer.intel.com/design/i960/papers/i2oarch.htm

[Cyclone]

[Gillespie]
B. Gillespie, "PCI Intelligent I/O Design for High Performance Servers", Intel Corp.,

[I2OSIG1]
I2O SIG, "I2O Questions and Answers", http://www.i2osig.org/QandA.html

[I2OSIG2]
I2O SIG, "I2O Technology Backgrounder", http://www.i2osig.org/TechBack.html

[I2OSIG3]

[Shanley]

[Wilner]
D. Wilner, "I2O IRTOS", Wind River Systems,
http://developer.intel.com/design/i960/papers/i2ortos.htm

[Intel1]

[Intel2]

[Intel3]
Intel Corporation, "I2O Initiative Management Overview",
http://developer.intel.com/design/i960/papers/27284501.htm

[Intel4]
Intel Corp., “i960 and I/O Innovation”,  
http://developer.intel.com/design/i960/INNOVAT.HTM

[Intel5]


[Intel6]


[Intel7]

Intel Corp., "Intelligent I/O in the PC-Based Enterprise Computing Environment",  
http://developer.intel.com/design/i960/applnots/bkgrnder.htm

[Intel8]


[Intel9]


[Intel10]


[Intel11]


[Intel12]

Appendix A : Source Code

/* test1.c
 * Test program for the IQ80960 Evaluation Platform.
 */

#include <stdio.h>

/* Address of Eight User-Programmable LEDs on IQ80960 board */
volatile char * LED_ADDR = (volatile char *)0xE0040000;

main()
{
    int i, j; /* Used as loop counters */
    int led; /* Store LED number here */

    *LED_ADDR = 0xFF; /* Turn LEDs OFF */

    /* Standard Input/Output routines are provided through MON960.
    * The output is sent to the debugger on the host using the i960's
    * message registers. The input is sent from the debugger on the host
    * to MON960 running on the target through the message registers.
    */

    printf("Testing the IQ80960 Evaluation Platform...
");
    printf("Press a key to see the LED's on the board blink...");

    getchar(); /* Wait for key press */
printf("The LEDs are now blinking...\n");

for(i = 0; i <= 40; i++) {
    *LED_ADDR = 0xFF; /*Turn LEDs OFF*/

    /* Delay */
    for(j = 0; j <= 50000; j++);

    *LED_ADDR = 0x00; /*Turn LEDs ON*/

    /* Delay */
    for(j = 0; j <= 50000; j++);
}

*LED_ADDR = 0xFF; /* Turn LEDs OFF */

while(1) {
    printf("Choose an LED to turn on (1-8, 0 to exit): ");

    scanf("%d", &led); /* Get led number from user */
    printf("%d\n", led); /* Echo user input */

    switch(led) {
    case 0 : /* Check for exit command */
        printf("Good Bye!\n");
        exit(0);
        break;

    case 8 :
        *LED_ADDR = 0x7F; /* Turn on 8th LED */
        break;

    case 7 :
*LED_ADDR = 0xBF; /* Turn on 7th LED */
break;

case 6 :
    *LED_ADDR = 0xDF; /* Turn on 6th LED */
    break;

case 5 :
    *LED_ADDR = 0xEF; /* Turn on 5th LED */
    break;

case 4 :
    *LED_ADDR = 0xF7; /* Turn on 4th LED */
    break;

case 3 :
    *LED_ADDR = 0xFB; /* Turn on 3rd LED */
    break;

case 2 :
    *LED_ADDR = 0xFD; /* Turn on 2nd LED */
    break;

case 1 :
    *LED_ADDR = 0xFE; /* Turn on 1st LED */
    break;

default :
    printf("Invalid Entry\n");
    break;

}
Appendix B : Disassembly Output

0xa00081d0 <main>:
  addo 16,sp,sp
0xa00081d4 <main+4>:
  mov g14,r3
0xa00081d8 <main+8>:
  lda 0xff <__profile_data_length+255>,g4
0xa00081dc <main+12>:
  st g4,0xa000f754 <LED_ADDR>
0xa00081e4 <main+20>:
  lda 0xa0008100 <_Btext+256>,g0
0xa00081ec <main+28>:
  call 0xa00087f0 <printf>
0xa00081f0 <main+32>:
  mov g0,g4
0xa00081f4 <main+36>:
  lda 0xa0008130 <_Btext+304>,g0
0xa00081fc <main+44>:
  call 0xa00087f0 <printf>
0xa0008200 <main+48>:
  mov g0,g4
0xa0008204 <main+52>:
  bal 0xa000f48c <stdio_ptr.lf>
0xa0008208 <main+56>:
  mov g0,g5
0xa000820c <main+60>:
  mov g5,g4
0xa0008210 <main+64>:
  ld 0x4 <__profile_data_length+4>(g4),g5
0xa0008214 <main+68>:
  subo 1,g5,g6
0xa0008218 <main+72>:
  st g6,0x4 <__profile_data_length+4>(g4)
0xa000821c <main+76>:
  cmpi g5,0
0xa0008220 <main+80>:
  ble 0xa0008240 <main+112>
0xa0008224 <main+84>:
  bal 0xa000f48c <stdio_ptr.lf>
0xa0008228 <main+88>:
  mov g0,g5
0xa000822c <main+92>:
  mov g5,g4
0xa0008230 <main+96>:
  ld (g4),g5
0xa0008234 <main+100>:
  addo g5,1,g6
0xa0008238 <main+104>:
  st g6,(g4)
0xa000823c <main+108>:
  b 0xa0008258 <main+136>
0xa0008240 <main+112>:
  bal 0xa000f48c <stdio_ptr.lf>
0xa0008244 <main+116>:
  mov g0,g4
0xa0008248 <main+120>:
  mov g4,g5
0xa000824c <main+124>: mov g5,g0
0xa0008250 <main+128>: call 0xa0008580 <_filbuf>
0xa0008254 <main+132>: mov g0,g4
0xa0008258 <main+136>: lda 0xa0008170 <_Btext+368>,g0
0xa0008260 <main+144>: call 0xa00087f0 <printf>
0xa0008264 <main+148>: mov g0,g4
0xa0008268 <main+152>: mov 0,g4
0xa000826c <main+156>: st g4,0x40 <__profile_data_length+64>(fp)
0xa0008270 <main+160>: ld 0x40 <__profile_data_length+64>(fp),g4
0xa0008274 <main+164>: addo 31,9,g5
0xa0008278 <main+168>: cmpi g4,g5
0xa000827c <main+172>: ble 0xa0008284 <main+180>
0xa0008280 <main+176>: b 0xa0008320 <main+336>
0xa0008284 <main+180>: ld 0x40 <__profile_data_length+64>(fp),g4
0xa0008288 <main+184>: subo 1,0,g5
0xa0008290 <main+192>: stob g5,(g4)
0xa0008294 <main+196>: mov 0,g4
0xa0008298 <main+200>: st g4,0x44 <__profile_data_length+68>(fp)
0xa000829c <main+204>: ld 0x44 <__profile_data_length+68>(fp),g4
0xa00082a0 <main+208>: lda 0xc350 <fpem_CA_AC+49744>,g5
0xa00082a4 <main+212>: cmpi g4,g5
0xa00082a8 <main+216>: ble 0xa00082b4 <main+228>
0xa00082ac <main+220>: b 0xa0008320 <main+336>
0xa00082b0 <main+224>: ld 0x44 <__profile_data_length+68>(fp),g4
0xa00082b4 <main+228>: addo g5,1,g4
0xa00082bc <main+236>: mov g4,g5
0xa00082c0 <main+240>: st g5,0x44 <__profile_data_length+68>(fp)
0xa00082c4 <main+244>: b 0xa000829c <main+204>
0xa00082c8 <main+248>: ld 0x40 <__profile_data_length+64>(fp),g4
0xa00082cc <main+252>: mov 0,g5
0xa00082d0 <main+256>: stob g5,(g4)
0xa00082d4 <main+260>: mov 0,g4
0xa00082dc <main+268>:   st   g4,0x44 <__profile_data_length+68>(fp)
0xa00082e0 <main+272>:   ld   0x44 <__profile_data_length+68>(fp),g4
0xa00082e4 <main+276>:   lda  0xc350 <fpem_CA_AC+49744>,g5
0xa00082ec <main+284>:   cmpi g4,g5
0xa00082f0 <main+288>:   ble  0xa00082f8 <main+296>
0xa00082f4 <main+292>:   b   0xa000830c <main+316>
0xa00082f8 <main+296>:   ld   0x44 <__profile_data_length+68>(fp),g5
0xa00082fc <main+300>:   addo g5,1,g4
0xa0008300 <main+304>:   mov g4,g5
0xa0008304 <main+308>:   st   g5,0x44 <__profile_data_length+68>(fp)
0xa0008308 <main+312>:   b   0xa00082e0 <main+272>
0xa000830c <main+316>:   ld   0x40 <__profile_data_length+64>(fp),g5
0xa0008310 <main+320>:   addo g5,1,g4
0xa0008314 <main+324>:   mov g4,g5
0xa0008318 <main+328>:   st   g5,0x40 <__profile_data_length+64>(fp)
0xa000831c <main+332>:   b   0xa0008270 <main+160>
0xa0008320 <main+336>:   lda  0xff <__profile_data_length+255>,g4
0xa0008324 <main+340>:   st   g4,0xa000f754 <LED_ADDR>
0xa000832c <main+348>:   b   0xa000833c <main+356>
0xa0008330 <main+352>:   b   0xa0008330 <main+352>
0xa0008334 <main+356>:   lda  0xa0008190 <_Btext+400>,g0
0xa000833c <main+364>:   call 0xa00087f0 <printf>
0xa0008340 <main+368>:   mov g0,g4
0xa0008344 <main+372>:   lda  0x48 <__profile_data_length+72>(fp),g4
0xa0008348 <main+376>:   lda  0xa00081bc <_Btext+444>,g0
0xa0008350 <main+384>:   mov g4,g1
0xa0008354 <main+388>:   call 0xa0008970 <scanf>
0xa0008358 <main+392>:   mov g0,g4
0xa000835c <main+396>:   ld   0x48 <__profile_data_length+72>(fp),g4
0xa0008360 <main+400>:   lda  0xffffffffd <stackbase+1610416576>,g5
0xa0008368 <main+408>:   addo g4,g5,g4
0xa000836c <main+412>:   cmpo g4,g8
0xa0008370 <main+416>: bg 0xa000845c <main+652>
0xa0008374 <main+420>: shlo 2,g4,g5
0xa0008378 <main+424>: lda 0xa000838c <main+444>,g6
0xa0008380 <main+432>: addo g5,g6,g4
0xa0008384 <main+436>: ld (g4),g5
0xa0008388 <main+440>: bx (g5)
0xa000838c <main+444>: ldt 0x3b0 <fpem_CA_AC+688>,pfp
0xa0008390 <main+448>: ldt 0x448 <fpem_CA_AC+840>,pfp
0xa0008394 <main+452>: ldt 0x434 <fpem_CA_AC+820>,pfp
0xa0008398 <main+456>: ldt 0x420 <fpem_CA_AC+800>,pfp
0xa000839c <main+460>: ldt 0x40c <fpem_CA_AC+780>,pfp
0xa00083a0 <main+464>: ldt 0x3f8 <fpem_CA_AC+760>,pfp
0xa00083a4 <main+468>: ldt 0x3e0 <fpem_CA_AC+736>,pfp
0xa00083a8 <main+472>: ldt 0x3c8 <fpem_CA_AC+712>,pfp
0xa00083ac <main+476>: ldt 0x3b4 <fpem_CA_AC+692>,pfp
0xa00083b0 <main+480>: b 0xa0008470 <main+672>
0xa00083b4 <main+484>: ld 0xa000f754 <LED_ADDR>,g4
0xa00083bc <main+492>: lda 0x7f <__profile_data_length+127>,g5
0xa00083c0 <main+496>: stob g5,(g4)
0xa00083c4 <main+500>: b 0xa0008470 <main+672>
0xa00083c8 <main+504>: ld 0xa000f754 <LED_ADDR>,g4
0xa00083d0 <main+512>: lda 0xffffffbf <stackbase+1610416559>,g5
0xa00083d8 <main+520>: stob g5,(g4)
0xa00083dc <main+524>: b 0xa0008470 <main+672>
0xa00083e0 <main+528>: ld 0xa000f754 <LED_ADDR>,g4
0xa00083e8 <main+536>: lda 0xffffffffdf <stackbase+1610416591>,g5
0xa00083f0 <main+544>: stob g5,(g4)
0xa00083f4 <main+548>: b 0xa0008470 <main+672>
0xa00083f8 <main+552>: ld 0xa000f754 <LED_ADDR>,g4
0xa0008400 <main+560>: subo 17,0,g5
0xa0008404 <main+564>: stob g5,(g4)
0xa0008408 <main+568>: b 0xa0008470 <main+672>
0xa000840c <main+572>: ld 0xa000f754 <LED_ADDR>,g4
0xa0008414 <main+580>: subo 9,0,g5
0xa0008418 <main+584>: stob g5,(g4)
0xa000841c <main+588>: b 0xa0008470 <main+672>
0xa0008420 <main+592>: ld 0xa000f754 <LED_ADDR>,g4
0xa0008428 <main+600>: subo 5,0,g5
0xa000842c <main+604>: stob g5,(g4)
0xa0008430 <main+608>: b 0xa0008470 <main+672>
0xa0008434 <main+612>: ld 0xa000f754 <LED_ADDR>,g4
0xa000843c <main+620>: subo 3,0,g5
0xa0008440 <main+624>: stob g5,(g4)
0xa0008444 <main+628>: b 0xa0008470 <main+672>
0xa0008448 <main+632>: ld 0xa000f754 <LED_ADDR>,g4
0xa0008450 <main+640>: subo 2,0,g5
0xa0008454 <main+644>: stob g5,(g4)
0xa0008458 <main+648>: b 0xa0008470 <main+672>
0xa000845c <main+652>: lda 0xa00081c0 <_Btext+448>,g0
0xa0008464 <main+660>: call 0xa00087f0 <printf>
0xa0008468 <main+664>: mov g0,g4
0xa000846c <main+668>: b 0xa0008470 <main+672>
0xa0008470 <main+672>: b 0xa000832c <main+348>
0xa0008474 <main+676>: ret

End of assembler dump.