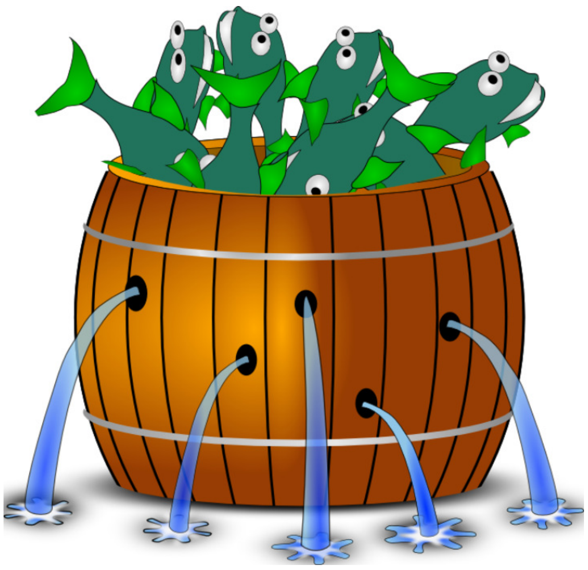
Barrelfish Project
ETH Zurich



Message Notifications

Barrelfish Technical Note 9

Barrelfish project

16.06.2010

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	16.06.2010	RI	Initial version

Chapter 1

Overview

1.1 Introduction

Inter-core messaging on Barrelfish (UMP) is currently based on shared memory circular buffers and a polling mechanism which is designed to work efficiently given the cache-coherence protocols of a typical NUMA multiprocessor system. Communication latency can vary by many orders of magnitude depending on how frequently the receiving process polls each channel. This document describes the design and implementation of a new kernel notification primitive for Barrelfish.

The reason I believe we need an IDC notification path can be seen in most of the traces I took of Tim's IDC and THC test program (see email of 21/5/10).

screenshot goes here

If you look on core 0 you see 3 domains polling for incoming URPC messages. Each polls for a while and then yields, and with 3 domains it takes up to 10000 cycles to notice a message, and obviously the current mechanism will scale with the number of domains on the destination core. (always ¿= 2!). This could be reduced by moving polling into the kernel, but if any domain is running we have no way to pre-empt it until the next timer interrupt (about 18 million cycles!).

1.1.1 Polling and cache coherence

Sending a message involves the sender modifying a single cache line which (in the expected case) the receiver is actively polling. The cache line starts in shared (S) mode in the cache of both sender and receiver cores. When the sender writes to the cache line this causes a transition to the owned (O) state and an invalidation of the copy in the receiver's cache. On most of our NUMA systems, be they Hypertransport, QPI or shared bus, this is effectively a system-wide broadcast. Newer AMD Istanbul processors have a directory-based cache coherency protocol which avoids the broadcast. The receiver then pulls the modified cache line from the sender's cache resulting in the cache line being in both caches in the shared (S) state.

1.1.2 Message Latency

When sender and receiver threads are the only things running on each core this can be extremely low latency (600 cycles). When the destination core is shared by multiple threads, or even multiple domains, the message latency is determined by kernel- and user-mode scheduling policies and is typically a function of the kernel clock interrupt rate and the number of domains (and channels) in the system. Even in simple cases the message latency will usually increase to at least one timer tick (at least 1ms, probably 10ms - i.e. millions of cycles!)

Several Barrelfish papers have talked about sechemes where receiver domains initially poll for messages, but eventually back off to a more heavyweight blocking mechanism. In the current tree this involves domains eventually "handing off" the polling of message channels to their local monitor process via a (blocking) local IPC. The monitor polls URPC channels for *all* blocked domains an when it finds a message it sends an IPC to the receiver process causing it to wake. Since all these cache lines will be "hot" in the cache, this is not as expensive as it might appear, but still does not allow for preemption of a running thread before the next clock interrupt. It also potentially captures kernel scheduling policy.

1.1.3 Scalability

Barrelfish does not currently multiplex URPC channels in any way, so it is common to see $O(N)$ and even $O(N^2)$ URPC channels between services which run on each core (e.g. monitors), or between the dispatchers of a domain which "spans" multiple cores. Though the memory consumption is not a huge problem (a URPC channel is a handful of cache lines), the number of channels can grow rapidly and this will have an effect on polling costs and message latency.

Domains do not have any efficient way to identify the (probably) small set of URPC channels which currently have pending messages. In Nemesis this was achieved by each domain having a "hint FIFO" which contained a list of channels with *new* incoming messages (identifying new messages required an explicit *acknowlegement count* in each channel). However, when a domain was activated it could efficiently dispatch new messages, and if the FIFO overflowed then the domain resorted to polling all channels.

In a many-core environment, a single hint FIFO would potentitally be an expensive bottleneck due to shared-write cache lines accessed by many cores. Having a FIFO for each potential sending core requires $O(N)$ space, but would avoid contention.

1.1.4 Primitives

Ideally we need primitives which allow fast URPC-style messaging when a receiver is known to be polling, but which also allow us to control sending timely notifications to a remote kernel, domain and thread.

Chapter 2

Design

In the global kernel data page have an array of pointers to per-core notification (PCN) pages. (by the way, why is struct global still declared in kputchar.h! :-) Each per-core notification page is divided into cacheline-sized slots. (i.e. 1 page has 64 slots of 64-bytes) There is one slot for each sending core. Each slot is treated as a shared FIFO with some agreed number of entries. Each entry can contain a short channel ID or zero. Each kernel keeps private arrays of head pointers and tail pointers (need only 1 byte per entry x num cpus).

On the sending side I have a new system call:

sys_notify(dest_core, dest_chanid);

This looks up the PCN page of the destination core in the kernel globals page. It then indexes into the PCN page using its own core id to locate the pairwise notification FIFO. It looks up the FIFO head pointer using the dest_core id. It then looks at that entry of the FIFO... if it is non-zero then the fifo is full. If the entry is zero then it writes the dest_chanid and increments the private head pointer.

On the receiving side, the destination core keeps a private index into each of its incoming fifos. These tells it which entry it needs to look at next. It could therefore poll each of the FIFOs waiting for a non-zero channelID value...

The mechanism so far results in a single FIFO cacheline toggling between Shared and Modified state on both sender and receiver. I timed 10000 invocations of the above sys_notify() call (with the receiver core in a tight polling loop) with a cost of 350 cycles per notification (for a shared L3) and 450 cycles cross-package. Note that the extra cache traffic of this design is probably not optimal, but it's in the noise compared to our current IDC costs when we have >1 domain on a core (i.e. always!)

Obviously a tight polling loop on the receiver is not ideal... we aren't always polling, and in any event this would scale as O(N cores).

One solution is for sys_notify() to send an IPI to dest_core whenever the FIFO goes non-empty, or at the request of the receiver (e.g. if it wrote a special 'request IPI' value into the next empty slot, rather than zero).

Given the number of CPUs in our current ccNUMA machines, we could easily afford to use a separate interrupt vector for each sending core. This would identify to the destination core which FIFO to look at, with no polling overhead. We could use a single IPI vector, but this would need some hierarchical shared datastructure to efficiently identify which fifos to poll. (the PCN entry for src_core == dest_core is unused and could be treated as 512 flag-bits c/f Simons RCK code)

Sending the IPI within sys_notify() would take a few hundred extra cycles (but may overlap with the cache coherence messages?) Taking the IRQ and acking it on the receiver is probably between 500 and 1000 cycles depending on which ring the destination core is executing in. (I tried Richard's HLT in Ring0 with interrupts disabled trick and it does not work any more!).

One interesting trick might be to deliver notifications to a hyperthread so that interrupt latency and polling costs were interleaved with normal processing...and only interrupt the 'application hyperthread' if a reschedule is necessary.

In all of the above cases, I would imagine that notifications would not necessarily cause the running domain to be pre-empted. However a scheduler activation for incoming IDC would be possible.

Given that true polling URPC *could* cost only a few hundred cycles if both domains are in a tight loop, this notification mechanism is not something I would imagine using on each message. Instead I would suggest having a 'PUSH' flag you can pass on urpc_send(), or a b-¿push() method on the flounder binding so the programmer can decide to expedite message delivery a suitable points.

Chapter 3

Implementation

I just finished doing a more complete version of the UMP Notification mechanism. It now uses a UmpNotify capability which you get by retyping your Dispatcher cap. The monitor's UMP binding mechanism already propagates a notification cap between client and server. I hand edited the bench.if stubs to allocate the caps and invoke the notification when doing a message send.

The cap_invoke handler puts the receivers's DCB pointer into the destination core's incoming notification FIFO and sends an IPI with a vector identifying the sender core (allowing demux without polling). The receiving core has an notification IPI handler which drains the notification fifo and does a cswitch to the most recently notified domain. The domain will get an activation and poll its URPC channels (eventually).

After thinking a bit more about the costs of notification, it seems that 2800 cycles is quite a lot to pay in the default path to send a notification. Quite a bit of this is the high cost of cap_invoke on x86_64 (2K cycles) compared to the hacky syscall I was using last week (800), but in general we don't want to pay much for notification unless it's necessary.

I added code in the domain dispatch path to publish the identity of the currently running DCB, and ideally the time at which it will next be preempted. The notification kernel code on the sender side can therefore tell if it's worth sending an IPI and return immediately if the domain is already running. This leads to the behaviour below where, just before T=70000 core 1 sends a message to core 2, notices it isn't the currently running domain and so sends a notification IPI. The monitor on core2 is preempted and the receiver domain gets to run. Activation code takes about 2000 cycles but the message gets there pretty quickly.

This allows me to increase MAX_POLLS_PER_TIMESLICE to 1000 without excessive penalty, which in turn allows the client and server to remain in the polling loop and notice messages before they yield to other domains. Net result is that the common case RPC cost is about 4x faster. The worst case is hopefully bounded by the cost of (cap_invoke + notifiy IPI + domain activation + ump_poll)... about 4000 cycles. I ought to check this by running some "while(1)" domains on each core... but I'm fairly (naively?) optimistic.

3.1 Performance

Chapter 4

Testing and Debugging