XS-1 Operating System Services (XOSS)

1 Introduction

This document is a project proposal for XS-1 Operating System Services (XOSS).

1.1 Aims

The aim of XOSS is twofold:

- to expand the operating system services provided by XS-1 cores and the XC language,
- to integrate with multi-processor operating systems and extend application parallelism.

1.2 Summary

An XMOS XS-1 is a multi-core processor, each core with its own memory and i/o links [1]. The cores embed support for a number of services commonly provided by an operating system on general purpose processors, including multi-threading, communication, general purpose i/o, and timers; and instruction set locks provide for semaphores in software while pseudo-dynamic services can be supported in XC with *select...case* statements. So, using XC, an operating system for XS-1 processors is not a necessity in an embedded homogenous system.

However, there are common operating system services which the XS-1 core does not embed, such as memory caching which would allow use of both a secondary cache and an external main memory, and services which XC does not support, such as a service registry which would allow a set of distributed services to be run across heterogenous processor networks. And the scope for parallel applications can be extended to run not just on multi-cores but also over multi-processors.

2 Expansion of XS-1 Services

While the XMOS architecture makes it practical to use software to perform many functions that are traditionally implemented in hardware, integration with popular multi-processor operating system software (e.g. VxWorks [4] and GNU/Linux[5]) could further open up established marketplaces and user bases to the new XMOS technology. Accordingly, those extensions under consideration with XOSS should facilitate integration with popular operating system software.

2.1 Secondary memory and Paging

A common memory hierarchy is depicted in Illustration 1:

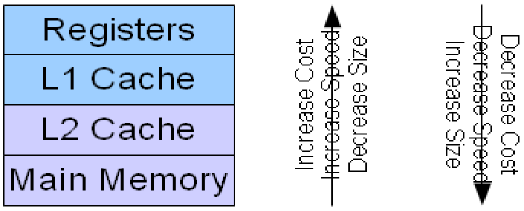

Illustration 1: Memory hierarchy

The XS-1 core register file resides at the top of the memory hierarhy; it provides the fastest access to data but is the smallest memory object. The XS-1 has 12 general purpose operand registers (not counting the 12 assigned registers from cp to ksp). Beneath the register file is the XS-1 per core 64KB on-chip memory, which is the next highest performance memory object and is accessed in a single-cycle; this can be considered a Level-1 Cache, although the XS-1 has no caching hardware.

The XS-1 provides no memory objects beneath the L1 Cache, and the processor has no ability to address external memory (though driving an external 100MHz SDRAM as a data storage device is straightforward). To provide the next level multi-cycle Level-2 Cache and lowest Main Memory necessitates both a system design and either toolchain support for XC or a custom software library.

2.1.1 Secondary memory system design

A proposed Level-2 Cache and Main Memory system design is depicted in Illustration 2:

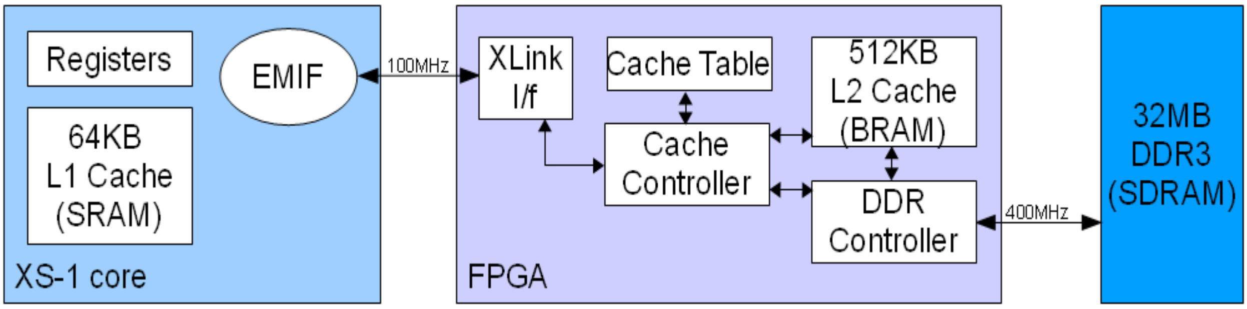

Illustration 2: L2-Cache and Main Memory system design

The XS-1 100MHz Xlink speed is programmable, but is assumed the optimum performance. And whereas the manufacture of slower speed SDRAM is being phased out, faster DDR3 memory at 400MHz is commonly available and is expected to have several years manufacturing lifetime. As the XS-1 cannot drive an interface at 400MHz, an FPGA can be used to interface DDR.

Using VHDL (or Verilog), a DDR Controller can be implemented to drive an external DDR SDRAM device, and an L2-Cache can be implemented directly in FPGA memory blocks called BlockRAM (BRAM), and a Cache Controller can be implemented to interface with an XS-1 core through an Xlink interface. (Multiple Xlink interfaces can support several XS-1 cores.) In software, XOSS can provide an External Memory Interface (EMIF) library that uses a protocol (XEP) for data exchange with the Cache Controller which maintains a Cache Table.

In one implementation the FPGA and external SDRAM can be located on an XStamp board (on which a SPI FLASH may also be located to store a bootable image). In another, the XMOS core IP could be implemented as an embedded processor within an FPGA [3]. The link between the EMIF and the Cache Controller could use the Xlink-on-FPGA provided by Paul Hampson and Ali Dixon.

2.1.2 Performance

The L2 Cache can be any size; illustrated is 64KB * 8 = 512KB which happens to equal 64KB times the number of XS-1 core threads. At 100MHz a 64KB transfer between the XS-1 core and the Cache Controller would take 5000µS; further, if the Cache Controller determines a transfer is needed from SDRAM through the DDR Controller into L2 Cache at 400MHz, then the transfer to the XS-1 core would total 6250µS. This is long enough to desire it infrequently or at known scheduling points, and moreso, as the XS-1 has no caching hardware, XOSS has to provide an efficient means of cache management.

2.1.3 Paging

Moving data between the register file and L1-Cache is a program function, using instructions like LDW, and it is the programmer's or compiler's responsibility to select an instruction sequence that keeps heavily referenced data in the registers as long as possible. However, a program is largely unaware of the memory hierarchy, and in particular cache access is generally transparent to it.

2.1.3.1 Toolchain support

One way to introduce cache management is to extend the XC software toolchain, providing the compiler with a way to 'signal' a cache-miss and the linker with a way to 'map' pages. Another way is to provide a library of functions with a C-language binding, and call them explicitly. However, an XC-based solution is explored as it is the XMOS Operating System Language.

2.1.3.2 Extended PAR

As noted above cache-misses would stall the processor for a large number of cycles, so facilitating them at known scheduling points is desirable. The XC programming language includes just such a known scheduling point in the PAR statement, where the programmer spawns threads. For example:

```
PAR
{
  a( );
  b( );
  c( );
}
```

causes the current thread to spawn two new threads which will run from the *a* and *b* entry points and to run from the *c* entry point itself. Only once all three threads *return*, or join, is the statement following the PAR executed (by the original current thread). Consider the following proposed extended PAR, in which the set of threads is named and declared with proposed new keywords:

```
PAR set_of_abc IS PAGED
{
  a( );
  b( );
  c( );
}
```

in which *set_of_abc* is a name like a variable name, and *IS PAGED* is an attribute. This works similarly for replicated PAR, for example:

```
PAR( int i=0; i<4; i++ ) array_of_a IS PAGED
{
  a[ i ]( );
}
```

2.1.3.3 Compiler and Linker

The proposed PAGED PAR gives the set of threads a name that can be passed to the linker, to layout a segmented memory map, and that can be enumerated for storage in the external Cache Controller Cache Table. It can cause the compiler to: reset the code and data address assignment to some 'page base' (allowing overlay of the same memory locations by other pages); and to call XOSS EMIF library code for communicating with the external Cache Controller, that will be executed in kernel mode [2] prior to the thread creation; and similarly to restore the former page at the join. Otherwise it does what a PAR does.

There are obvious limits nesting PAGED PAR within an outer PAR, that the compiler can check in addition to limits on interrupts or channel i/o. Compilation fixes the (global) constant pool (cp) and data pointer (dp) registers in memory above the 'page base'; and context management (sp) needs space, so that only on-chip memory (L1-Cache) below the 'page base' is swapped.

2.1.4 Bootstrap and Loader

Paging out of on-chip SRAM (L1-Cache) through the FPGA into L2-Cache or onto SDRAM requires no further preparation, but before code or data which hasn't previously been swapped out (e.g. *set_of_abc* on first entry to the PAGED PAR) can be paged into the XS-1 SRAM from L2-Cache or SDRAM, it needs to have been placed there. Pre-placement needs to be done at boot time, so a binary image that contains page segments requires a secondary bootloader (linked with the XOSS EMIF library) to pre-load each segment into L2-Cache and populate the Cache Table, before loading the image main entry page into on-chip RAM and jumping to it. If during pre-placement the L2-Cache is filled, the Cache Controller has to push pages onto SDRAM.

2.1.5 Exceptions

The XOSS EMIF library code needs to detect the presence of an external Cache Controller, through the XEP, and raise a kernel exception [2] if it fails to communicate.

2.2 Common Services

As XS-1 on-chip memory is finite XOSS should maintain a small memory footprint, and so only a minimal set of common services should be provided, enough to integrate with external multi-processor operating systems.

2.2.1 Service Registry

Each service in an XS-1 network provides a published capability, for example a device driver for a temperature sensor, which application processes reach through channels.

2.2.1.1 Dynamic services and Channels

While XC or C provide for pseudo-dynamic services through a *switch* based on some initial *case* token, selecting 1-from-N set of services to run in an XS-1 thread, provision of runtime paging brings about the possibility to dynamically load code onto an Xcore and thus to spawn truly dynamic services, either with or without pre-allocated channel endpoints.

Allocating channels is typically shielded by the XC compiler for static allocation at compile time. Allocation of channel resources to networks and threads is done at the machine-instruction level [1]. XOSS software libraries can be provided to ease application development for dynamic channel allocation. Each location in an Xcore network is addressed using a triple `<chip-address, core-address, channel-address>`, with values assigned in each project XN file. The chip and core addresses are retrieved through the XC Library function `get_core_id()`; and the channel address is assigned when the resource is allocated, through a `getr` machine instruction.

2.2.1.2 Service location

The implementation of a XOSS Service Registry (XSR) can resemble an internet Domain Name Service (DNS), in which a look-up table relates a service name with a network location. On startup a service can advertise its presence through the XSR, allowing application processes to attach dynamically. The XSR process itself needs to run at a well-known network address, for example `<0, 0, 0>`, so that a dynamic service knows where to send messages for registration; this might be aided by creating an XSR entry in the XN file.

2.2.2 Virtual Channels

As each XS-1 core has a finite set of channel resources, XOSS could provide a library of Virtual Channels (VCH). A large number of VCHs could be multiplexed over a few physical channels.

2.2.3 Timer services

As each XS-1 core has a finite set of timer resources, XOSS could provide a software library of Virtual Timers (VTM). A large number of VTMs could be multiplexed onto a few physical timers.

2.2.4 Semaphore services

XOSS could provide a software library of semaphores (SEM), implemented using XS-1 locks. As each XS-1 core has a finite set of lock resources, a large number of SEMs could be multiplexed onto a fewer number of physical lock resources.

3 Integration with multi-processor OS

XOSS aims to integrate XS-1 multi-core networks with multi-processing operating system software. In one direction this makes an XS-1 network available as a slave co-processor array to multi-processor operating systems for performing specialist functions, and in the opposite direction it opens up host services (like file storage) to XS-1 applications. In addition to distributed services spread over a heterogenous network, the scope for developing parallel applications to run not just on multi-cores but also over multi-processors is expanded.

One possible multi-processing arrangement is shown in Illustration 3.

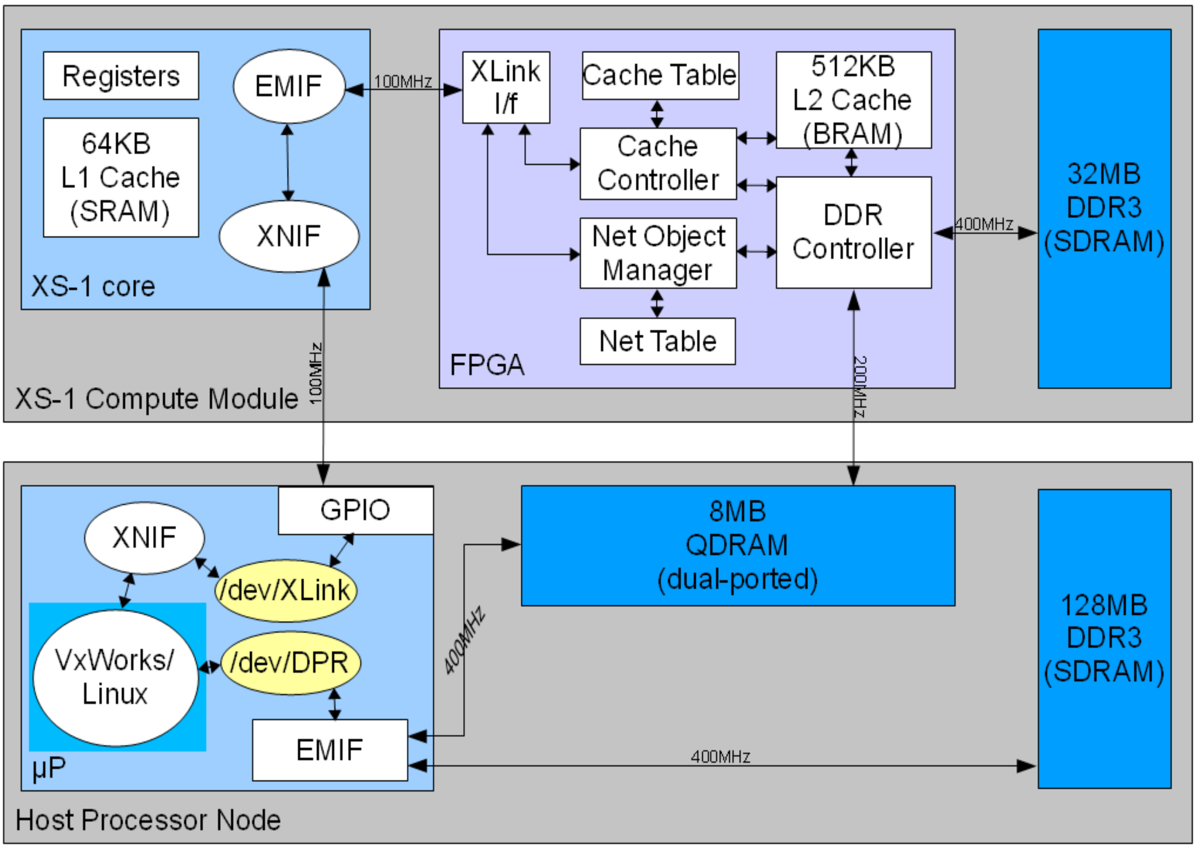

Illustration 3: Multi-processor System Design

A second Xlink clocked at 100MHz from the XS-1 would be used to attach with the Host Processor Node (HPN) GPIO pins. This would be partitioned (with respect to the XS-1 multi-cores) into an independent XOSS network, XNET. The XS-1 Compute Module (XCOM) FPGA is expanded to include network components, extending the DDR Controller with a DPRAM interface at 200MHz to share a dual-ported RAM [9] with the HPN; and a Net Table, used to map network object locations, enables the Network Object Manager to command the DPRAM interface when reading or writing those objects. (If a QDDR proves too large or costly, a DP-SRAM interface can be provided in the FPGA instead.) The HPN is assumed to have a local SDRAM.

Service extensions proposed in XOSS aim to integrate with multi-processing operating system software, like the real-time VxWorks [4] or the general purpose GNU/Linux [5]. Such kernels would not be run directly on XS-1 cores, as they are too large (μCLinux included). Instead, XOSS/XS-1 threads would interface with a host processor running those kernels, through heterogenous multi-processor networks.

A host-resident `/dev/Xlink` software device driver shall be provided for vxWorks or Linux to ease host application development of inter-processor communication over XNET/GPIO, and a XOSS Network Interface (XNIF) software library shall provide a protocol. The HPN shall use this protocol to command the XCOM, to load or exchange data with XOSS/XS-1 threads, and XNIF will allow XOSS/XS-1 threads access to host system services. A host-resident `/dev/DPR` software device driver shall be provided for the HPN to load data or code modules into DPRAM, prior to signalling the XCOM using XNIF, and for the HPN to collect computational results or pre-processed data, once signalled by the XCOM.

3.1 Hypervisors

In heterogenous asymmetric multi-processing systems, such as shown in Illustration 3, seperation between cores is distinct, whereas in homogenous symmetric multi-core systems, such as XS-1 arrays, the level of resource sharing between processors is higher.

A modern way to share computing resources is through the use of a software Hypervisor [6], which virtualises certain aspects of a system (i.e. processors, memory and devices) allowing distinct multiple operating systems to run concurrrently on top of it. There are two distinct types of Hypervisor: type 1 runs directly on and controls the hardware, to govern guest operating systems that run on another level above it; and type 2 is software application, running within a conventional operating system environment, as a distinct layer or library above which guest operating systems run at a third level making hypercalls into it.

3.1.1 WindRiver

WindRiver Hypervisor [7] is a type 1 hypervisor able to run in several distinct modes, and aims to share resources between different operating systems in multi-core and multi-processors systems. In one mode, called supervised Asymmetric Multi-Processing (sAPM) [7], the processors are not shared (or time-sliced) between operating systems, but both memory and devices are shared and protected by the Hypervisor. In this mode WindRiver Hypervisor in essence comprises a set of device drivers and Interrupt Service Routines (ISRs) for certain devices, e.g. `/dev/dpram`. It is this mode that most closely resembles Illustration 3, and the one XOSS shall target for inter-operation; vxWorks (and/or WindRiver Linux) will be able to run on the HPN atop WindRiver Hypervisor, and XOSS/XC/XS-1 shall run on the XCOM, sharing memory and devices over the XNET.

3.1.2 GNU/Linux

Interest in Linux as a hypervisor is growing [8]; KVM is a popular type 1 hypervisor, and QEMU and WINE are popular type 2 ones. In addition to providing a hypercall layer similar to kernel system calls, a core element of a Linux Hypervisor is a page mapper which points to memory pages used by the guest operating system(s) [8, figure 3]. It would be possible to go through `/dev/dpram` for storing memory pages, and to provide functions in XNIF for interfacing with XOSS/XS-1 applications. And the Linux Hypervisor can virtualise I/O, so exposing XOSS VCHs as a Linux device on the HPN would allow a direct exchange of data between XOSS/XS-1 and Linux threads.

Resembling Lguest [8] rather than KVM [8], XNET shall be provisioned with device drivers to interface an XS-1 network to a GNU/Linux host, and a host-resident suite of tools and libraries will allow programmatic control of the XS-1 sub-network. It may further be possible for host-resident kernel module(s) to present the XS-1 sub-network as a slave processor to Linux `/proc`. A suite of XCOM-resident libraries would allow XOSS/XS-1 threads access to host services such as backing storage.

Conclusion

Though XOSS 0.1 remains drafty and consequently full of holes, it will become more polished as we continue to learn about the proposed caching method, use of XN files, and Hypervisors. A case for integrating XC/XS-1 with popular multi-processor operating systems is asserted, which would benefit from expansion to the operating system services provided by XS-1 cores and the XC language (whether to XC or through a custom C library) and attention given to the host interface. Operating system hypervisors (in supervisor mode) provide for a coming-togetherness of heterogenous processor networks or arrays, and provision of Virtual Channels would expand the scope for parallel application development across multi-processor systems. Related areas of interest include IP-tunnelling through Xlinks and Virtual Channels.

References

[1] XMOS XS1 Architecture, May D., 2008

[2] XMOS XS1 Architecture Tutorial, May D., Mueller H., 2009

[3] FPGA Embedded Processors, Fletcher B., 2005

[4] VxWorks Operating System, Intel WindRiver

[5] Linux Operating System and also GNU/Linux

[6] Hypervisors, wikipedia

[7] Multicore challenges and choices, Evensen T., WindRiver, 2009

[8] Linux Hypervisor, Jones M.T., 2009

[9] Dual-ported DDR memory, IDT, 2009