Design Note: Target Considerations for Coldfire[†]

Jonathan S. Shapiro, Ph.D.
The EROS Group, LLC

March 15, 2006

Abstract

Documentation of gross issues and considerations in porting to the ColdFire V4e platform.

1 Introduction

This document addresses design considerations in porting the Coyotos kernel [1] to the ColdFire V4e processor family. This note specifically considers issues in porting to the MCF5485, but most of the issues raised and their solutions should be common to other members of the family.

The ColdFire V4e family generically meets the functional requirements to run Coyotos. It implements a paged memory management unit and suitable user vs. supervisor protection isolation. Privileged registers can be configured for supervisor-only access. Handling of sensitive registers is less careful, but these registers primarily disclose boot-time configuration information; there do not appear to be any substantive disclosures of interest to applications through these registers.

With these issues addressed, the following issues remain to be considered:

1. Feasibility of family compatibility with a single kernel.
2. Selection of Coyotos kernel-defined page size for this processor family.
3. Memory management subsystem and possible implementations of the Coyotos shadow translation mechanism for this architecture.
4. Design of processor cache(s), interactions with the memory management mechanism, and any required special handling for these.
5. Interrupt and trap frame formation, system call entry/exit mechanism(s).

2 Family Compatibility

Different members of the V4e family place certain key configuration registers at incompatible memory addresses. This has been an abiding problem for the GNU binutils maintainers, the problem has been explored fairly carefully. The current assemblers can be used to generate implementation-specific code where required, but there is no reason to believe that the Freescale team views forward supervisor-mode compatibility as a design objective.

Open Issue There is no published algorithm for performing CPU model identification within the V4e family. It is likely that something can be worked out, but since it isn't a pressing issue for any current users of this family, there are no immediate plans to investigate this further.

3 Page Size Selection

The ColdFire architecture supports a range of page sizes. The Coyotos kernel architecture is designed in terms of a single least page size that forms the atomic unit of address space construction. On the ColdFire, the feasible page size choices are 1K, 4K, or 8K. For this implementation, we have chosen 4K pages:

- Our intuition is that the 1K page size is not motivated for this processor.

  Current devices built around this family have reasonably large memories. The use of a 1K page size increases TLB pressure and cache line antialiasing challenges, and seems unlikely to result in a significant improvement in memory utilization in practice.

- This choice matches the IA-32 choice, and therefore minimizes systemic development risk.
- The 8K option seems likely to lead to underutilization of memory. Several processors that have previously selected an 8K page size have later been driven to subpage permission masks, either for reasons of compatibility or efficiency.

The primary concern arising from the 4K page size choice is cache coherency issues in the virtually indexed cache. This appears to be manageable through page coloring (discussed below).

Notwithstanding the general page size decision, it may be advantageous to use 8K or even 1M mappings to describe the kernel region in order to reduce kernel TLB overhead.

4 Memory Management Subsystem

The V4e family implements separate soft-loaded TLBs for code and data references. Each TLB has 32 entries and is fully associative. Entries can be individually pinned in the TLB, but there are no plans to exploit this in the initial port except as may be required to support the software translation architecture. Future implementations may exploit this feature more aggressively.

The use of a split I/D TLB architecture lets us enforce the Coyotos NX permission bit in software.

The soft-loaded architecture implies that most kernel virtual addresses can be reloaded without any lookup. Aside from the transient mapping region, the kernel portion of map can be configured as Virtual=Physical+constant. This permits very fast reload of supervisor-mode addresses following a simple bounds check.

The soft-loaded architecture also gives us an efficient way to *unmap* temporary mappings. The main challenge we will face with temporary mappings is the need to honor map colorings within the fast path.

Interim Translation Strategy Given the decision to use a 4K page size, it may actually make sense to adopt the IA-32 mapping logic unmodified, simulating the IA-32 page table search in the software miss handler. It's a pretty deficient way to handle translation on this architecture, but it would let us do a faster initial bringup. My main concern isn't so much the translation mechanism itself as the need to rethink the dependency tracking mechanism appropriate to a soft TLB.

Production Translation Strategy The most obvious strategy for this type of architecture is to implement a second-level TLB in software backed by a more conventional translation architecture. The soft TLB can be stored in KRAM for high-performance access. A comparable approach was successful in the R4000 (maybe it was R3000?) era kernels at Silicon Graphics. More recent processors have used soft-loaded TLBs with some degree of instruction set support as well. This strategy has the advantage that it can be mapped pretty easily to the existing dependency management logic.

That said, my sense is that this isn't the best approach to use. Instead, I think we should consider caching partial PATT traversals. In particular, we can build a cache of translations from translation roots up to but not including the leafmost PATT. At page fault time, we can consult this cache to quickly find the leaf PATT and then search that PATT to find the desired target of the mapping. That mapping can be directly loaded into the TLB from software subject to coloring constraints.

The idea of a traversal cache may be less pressing in Coyotos than in EROS, and the first thing to do is certainly to implement a fast soft traversal directly.

5 Caching Issues

The V4e family cache is virtually indexed and physically tagged. On the MCF5485 the set size is 8K. Given a 4K page size, this creates some messy coherency issues if virtual and physical addresses do not match in the least 13 bits.

The current proposal is to keep track of the active mapping colors for each physical frame. As long as all mappings are read-only, simply ignore the incoherency. The instant that a writable mapping exists, invalidate all mappings of non-matching color and hand-flush the relevant parts of the cache.

The implementation is a nuisance, but the practical performance implications of this are expected to be small:

- Shared code pages are never observed to be written in the wild.
- Purely read-only pages are not impacted.
- Private writable pages are impacted only if there are simultaneous non-congruent mappings. This is not observed in the wild.
- Shared pages are almost always mapped with compatibly congruent virtual addresses in order to gain the benefit of mapping metadata sharing. The practical consequence of this is that shared *writable* pages almost always turn out to have a single color in practice, and in this case the cache flush can be avoided.

If we observe that the coloring logic is actually getting triggered in practice, we may want to configure the cache in write-through mode.

When you are done wretching over this, consider that it could be worse: on the ARM it is necessary to guarantee that a writable page has exactly one mapping at a time, and further necessary to flush the entire cache when that mapping must be changed. On ColdFire we only need to flush *half* the cache. Oh frabjous day.[1]

Open Issue Section 5.2.3.5 of the MCF5485 reference manual [2] states that the cache uses the virtual address for indexing but the physical address for allocation, and notes that multiple mappings may lead to incoherency in the cache. However it appears that the incoherency can only arise in non-interacting cache sets, so this may be acceptable if no writes are possible.

However, it is possible that this statement is entirely literal. In that case, we will really need to guarantee that bit 12 must really match in the virtual and physical addresses at all times. If so, then attempts to map at a miscolored virtual address may require that we "bounce" the physical page to a compatible physical frame. This would be inconvenient, but it's doable. For the reasons given above it is unlikely to arise in practice, but it would be a shame if it actually proves to be necessary.

6 Interrupt and Trap Frames, System Calls

The V4e family implements a common exception frame for all interrupts, traps, and system calls. In contrast to IA-32, Interrupt and trap vectors do not overlap in the vector table. There is no specialized SYSCALL instruction. System calls are accomplished using the volitional TRAP instruction, which can dispatch to any of 16 trap vectors. All of these cause transition to supervisor mode and switch to a supervisor stack pointer. The regularity of the trap interface means that no system call entry or exit trampoline is required for this architecture.

One misfortune is that the TRAP instruction does *not* disable interrupts. This may preclude a fast register save design within the kernel as was done in the IA-32 implementation. The shortest path to disable appears to be something like:

```
mov  %sr,$disable
... proceed with save ...
```

[1] Read *Through the Looking Glass*. Do not pass go. Do not pause to admire yourself in the mirror. If you happen to run into an odd, heavyset, self-absorbed hare, say hello for me.

The problem here is that the fast save strategy requires that the kernel-mode stack on trap points directly into the per-process save area. This save area is fairly small, which presents two difficulties:

- Prior to the disabling of interrupts, we may take a hardware interrupt, which places an exception frame on the stack and tries to run the handler. To handle this, it will be necessary for these handlers to recognize having interrupted kernel mode in the assembly stub and avoid using more than a few words of the active kernel stack.
- For real entertainment, we might also take a TLB miss from either the initial path or the interrupt handler. This might lead to *another* exception frame and an attempt to run the reload handler while using the save area as a stack as well. It *may* be possible to constrain the register use of the TLB miss handler in this case as well, but we need to look into this pretty carefully.

My intuition is that it is worth it, and that we can pull it off, but we need to go counting stack words to make sure that this can all be pulled off. We will certainly need to pin the kernel stack mapping and the mapping for the fault handler code itself into the D-TLB and I-TLB.

In the interim, the simplest thing is to run off of the *real* kernel stack from the beginning, save a scratch register there temporarily, load the save area pointer into this register, and then use it as a base register to save the balance of the user-mode register set. The trampoline register can then be reloaded and saved to its proper location.

On this architecture, the only privileged state in the integer save area is the SR register state, which needs to be handled specially during reload in any case. The kernel interface no longer includes any operations that copy data to or from user space during kernel invocations *other than* IPC, which is an assembly path. Taken together, these facts may make it possible to entirely eliminate the savearea structure and reference all user-mode arguments directly from the UPCB structure on this architecture. For cache locality and security reasons I am reluctant to eliminate the savearea structure.

References

[1] J. S. Shapiro, Eric Northup, M. Scott Doerrie, and Swaroop Sridhar. *Coyotos Microkernel Specification*, 2006, available online at www.coyotos.org.

[2] Freescale, Inc. *MCF548x Reference Manual*, Jan 2006, Document Number MCF5485RM, Revision 3.