2nd Workshop on New Directions in Computer Architecture (NDCA−2), June, 2011

Memory Contexts: Supporting Selectable Cache and TLB Contexts

Tim Brecht,
David R. Cheriton School of Computer Science,
University of Waterloo
brecht@cs.uwaterloo.ca

ABSTRACT

In this paper I argue that in addition to supporting multiple cores, future microprocessor designs should decouple cores from caches and TLBs and support multiple, run-time selectable hardware memory contexts per core. Multiple memory contexts would allow the operating system and other threads to run without polluting each others' cache and TLB contexts, reduce coherence traffic, and enable better informed scheduling decisions, thus reducing execution times. In addition, it add provide significant flexibilty and benefits to virtualized environments.

1. MOTIVATION

An operating system captures a thread's execution context in a thread (or process) abstraction. This includes the contents of the processor's registers at the time of execution and a pointer to the stack used by that thread. While this abstraction was perfectly adequate when it was invented and it is still necessary to save and restore this part of a thread's context in order to resume execution at a later time or on another core, it does not sufficiently capture the remaining context of a thread's execution. Namely it does not allow the operating system to keep track the potentially large amounts of data that have been accumulated in the various levels of a processors caches and TLBs.

When a thread, T_1 runs on core C_1, it usually builds considerable context on that core. This consists of the contents of the core's registers, L1, L2, and in some cases a shared L3 cache. In addition the TLB may contain a large number of entries for that address space. When T_1 blocks (e.g., due to a page fault or blocking operating system call) a mode switch occurs to execute the operating system. While servicing the system call or page fault the operating system destroys significant cache and possibly TLB context [7]; it then dispatches a new thread T_2 to that core (C_1) which destroys even more of T_1's context. With a TLB that doesn't support address space identifiers, if T_2 belongs to a different address space than T_1, the entire TLB will need to be flushed when switching to T_2, completely destroying all of the TLB context. Eventually, when the resource that T_1 was blocked on is available, the operating system must decide which core to use to run T_1. This is a decision that it is ill suited to make because it has very little information available. The best choice of cores depends on how much of T_1's context remains on C_1, how much context T_2 has accumulated on C_1, how much of that context will be accessed in the future and how much context T_1 and T_2 have remaining on other cores they have run on. This proposal significantly improves the information available to the operating system, because it allows the operating system to associate memory contexts with threads.

2. MEMORY CONTEXTS

I define a *memory context* (an MC) to be an operating system selectable set of caches along with a selectable *TLB segment set* (defined later). This decouples caches and TLBs from cores and gives the operating system the ability to choose which MC to use with each core during execution of a thread. Ideally, an MC consists of multiple cache levels but latency constraints may require the L1 cache to be excluded (i.e., it may need to be bound to the core). Prior to dispatching a thread on a core, the operating system would program a set of registers in the core's MMU to choose the combination of caches and TLBs that is best suited to executing that thread. The operating system could specify that core C_i use cache set c_j and TLB set t_k (all levels in each case). If sufficient interconnect bandwidth could be provided and latencies are low enough, different components of a set could be chosen. For example, core C_i uses $L1_x$, $L2_y$, and $L3_z$. This could include possibly sharing a physical cache with another thread running on a different core (e.g., L3). A processor should contain many more memory contexts than cores, allowing the operating system to easily reuse contexts accumulated by threads on the same or a different core. If possible all contexts should be available to all cores (making thread migration trivial and efficient), but practical considerations may require having a limited number of MCs private to each core.

3. TLB SEGMENT SETS

The motivation for this part of the proposal comes from the fact that TLBs are relatively small and they typically don't cover the data that resides in all levels of a processor cache. With a first level TLB of 128 entries, a second level TLB of 512 entires and a 4 KB page size, a TLB can

only cover 2.5 MB. This is only a small portion of modern cache sizes that are often 8 – 24 MB or greater. The problem is exacerbated because with 64-bit address spaces and larger applications that access increasing amounts of data, operating systems often represent a program's address space using multi-level page tables. As a result, a TLB miss requires a page table walk that must access each level of the page table. If the page table walk misses in the cache, several memory accesses may be required to load the TLB. This is despite the data being available in the processor cache. I believe that this proposal will improve TLB coverage in addition to increasing the possibility that the page table data remains in a cache the kernel can access.

I propose extending first level TLB coverage by expanding the notion of separate instruction and data TLBs to include more address space segments. Separate TLBs could be used for code, data, heap, shared memory, shared library, and stack segments forming what I will call a *TLB segment set*, or TLBSS. Each TLB in the set is responsible for a different range of contiguous addresses that define the segment. During an address space switch, the operating system first selects the desired MC and then loads a pair of registers in each of the TLB segements in that segment set with the minimum and maximum addresses that define each segment. Not all TLBs in a TLBSS are required to be active and this would also be specified by the operating system. During address translation the core's MMU uses the specified MC and TLBSS. If the address being translated is outside of the range of all segment address register pairs, an exception is generated. The idea of supporting a TLBSS is independent of and does not require supporting multiple memory contexts.

Ultimate flexibility would be provided to the operating system by simply requiring the first and last virtual page to be specified for each TLB segment. However, it may not be possible to implement such an approach as efficiently as required for first level TLB accesses. Another approach that may be somewhat more efficient would be to use the high order 3 or 4 bits of the virtual address to specify predefined hard coded regions of virtual address space. This would require the compiler, linker, and operating system to use predefined ranges of addresses for each segment. While this would likely prove too restrictive for use in 32-bit machines, dividing a 64-bit address space into 8 or 16 segments each translated by a different TLB would be unlikely to cause problems for applications.

4. SOME RELATED WORK

This proposal is similar to previous work that partitions shared caches so that different threads or different types of data access do not destroy each others cache context. A few examples of such work include, [1], [9], [6] [8], [10]. Each of these previous studies have shown that several applications can acheive significantly reduced execution times by reducing cache misses. I believe that these studies provide evidence that this proposal could reduce execution times if it could be implemented efficiently.

However, in contrast to previous work, this proposal applies to all levels of the cache and also to TLBs. Instead of partitioning shared caches amount executing threads or cores, I hope that chip designers could instead replicate caches to provide a greater total amount of cache than is currently available but partition it in a way that permits access times that are equal to or only marginally greater than current systems.

Soares and Stumm[7] use some simple benchmarks to demonstrate the direct and indirect costs of mode switches. By simply entering and exiting the kernel with different frequencies, they show that the IPC is greatly reduced as the frequency of mode switches increases. Their work shows suprisingly high overheads due to simply trapping into and returning from the kernel.

Nellans *et al.* [5] are also concerned with the impact that kernel execution can have on application execution. They propose using a special cache per core that is used while executing in priveledged (kernel) mode. This prevents the operating system from destroying the context of the thread that was executing prior to the kernel, and that thread from destroying the kernel's context. Using simuilations they find that their approach can improve the IPC of operating system intensive applications by 18% to 55%.

This proposal is more general because the operating system is free to choose whether the kernel does or does not share caches with user threads and whether or not other threads share caches with each other. This allows for the possibility that the kernel (or another user thread) may be able to augment a thread's execution context (e.g., by touching data that the thread will be accessing [3], [2]). It also allows the kernel to prevent user-level threads from destroying each others' contexts. I believe that this work also provides evidence that significant benefits may result from providing the operating system with the ability to choose memory contexts.

Finally, Mogul *et al.* [4] argue that there is a growing gap between what operating system and computer architecture researchers think is important. They suggest that more communication should occur between these two groups and one of their proposals for future architectures is that they provide for software controlled cache management. This proposal is a modest step in the direction of both of these issues, it is a means of communicating to architecture researchers some ideas that I think could be useful for an operating system and describes one possible approach to cache management.

5. DISCUSSION

One of the driving forces behind these ideas is to significantly increase the amount of TLB and cache available in the system without significantly increasing latencies. In fact I believe that many applications would be better off if we moved to this type of caching arrangement rather than simply continuing to add cores. In contrast to previous work which takes existing shared cache designs and attempts to partition them among executing threads or cores, I'm proposing that existing TLB and cache hierarchies are replicated in such a

way that they can be selected.

The big question is whether or not these ideas could be implemented in such a way that the size of the total cache and TLB available would be significantly larger than existing systems but with latencies that are sufficiently low to be beneficial.

One possibility might be to be implement a low latency high bandwidth switch between cores and contexts. The hope is that the cost to select the desired memory context would be incurred at the time of a mode switch and/or context switch (which happens relatively infrequently relative to loads and stores). This could essentially create a virtual circuit between the core and the selected memory context. The hope is that after the memory context has been selected, additional overheads to route requests to and from that context would be small enough to make this approach worthwhile.

Besides the issues related to a possible core-to-cache network, it's latencies, and bandwidth, other interesting areas of research would include:

- What are the costs versus the benefits of increased delays in order to provide memory contexts? What levels of caches and TLBs can be decoupled with low enough latencies to still provide benefits. As mentioned previously, level one caches and TLBs may need to be tightly coupled with cores with perhaps decoupling implemented at higher levels where additional latencies would be a small fraction of existing latencies.

- What is a good number of segments to support in a TLB segment set? Can it be implemented in such a way that level 1 TLB coverage can be increased without significantly increasing delays.

- What are the costs and benefits of providing flexibility in choosing memory contexts? The most flexible approach would be to permit any L1, L2, L3 and any segments from any of the TLB segment sets to be chosen from (associated with) any core. However, this would require a relatively large network and switching mechanism and as a result it is likely the approach that would incur the highest latencies. Less flexible approaches may still provide significant benefits but

- How much flexiblity is required in order to still obtain benefits?

- What is a good number of memory contexts to support? This is likely to vary with the workload and perhaps processors could be sold with different numbers of cores and memory contexts. Once all memory contexts have been utilized, the operating system needs to decide which contexts to share between threads (assuming threads that were using those contexts are still alive).

- One of the key questions is are applications better off with a greater number of partitioned caches or just larger shared caches. As noted earlier, the hope is that an implementation could be provided that permits the memory context selection to essentially set up virtual circuits, thus permitting subsequent accesses to incur only minimal additional delays when compared with current processors.

I expect that it would also be beneficial if the operating system could specify caches that belong to a coherence set (i.e., the set of caches among which coherence is to be maintained). Because an inactive memory context may not require coherence with active memory contexts, it may present opportunities for more efficient and possibly lazily implementation (coherence is only needed when the inactive memory context is activated).

Increased flexibility could also be provided by implementing different replacement algorithms in different subsets of contexts, this would allow the most suitable algorithms to be selected for some applications.

6. EXPECTED BENEFITS

The expected benefits of this approach are:

- Significant reductions in cache and TLB pollution and thus execution time.

- Improved ability for a helper thread [3] [2] or the operating system to prefetch cache and/or TLB context for another thread. For example, when executing disk reading code the kernel might select an L3 data cache that is being used by the thread that will ultimately process the data being read from disk.

- Simplification of the operating system's scheduling decisions. With this proposal, each core is separated from its cache and TBL context and the operating system controls whether threads share memory contexts and what parts are shared (or are not shared).

- Increased TLB coverage (from TLB segment sets).

- The potential for reduced cache coherence traffic.

- TLB shootdown may be done on inactive TLBs without an inter-processor interrupt. For example, an inactive memory context may not require coherence with active memory contexts, or if coherence is required, it may be more efficient to implement this lazily (i.e., the next time the memory context is activated).

- Faster execution of operating systems and applications in virtualized environments. Virtual machines could each "own" a subset of memory contexts for them to manage. This would permit virtual machines to execute without destroying each others context as well as threads and kernel code within each virtual machine.

Acknowledgements

I would like to thank the members of the LabOS group at EPFL for several interesting discussions related to trying to understand and capture a more complete notion of a thread's context within an operating system. This work has been supported by the Natural Sciences and Engineering Research Council of Canada.

7. REFERENCES

[1] B. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, 1994.

[2] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[3] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for PreExecution. In *Proceedings of Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[4] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares. Mind the gap: Reconnecting architecture and os research. In *In Proceedings of Hot Topics on Operating Systems, XIII*, 2011.

[5] D. Nellans, R. Balasubramonian, and E. Brunvand. Interference aware cache designs for operating system execution. Technical Report UUCS-09-002, School of Computing University of Utah, 2009.

[6] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, 2006.

[7] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[8] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 258–269, 2008.

[9] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[10] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 89–102, 2009.