Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores

Vivy Suhendra, Tulika Mitra
School of Computing
National University of Singapore
{vivy, tulika}@comp.nus.edu.sg

ABSTRACT

Multi-core architectures consisting of multiple processing cores on a chip have become increasingly prevalent. Synthesizing hard real-time applications onto these platforms is quite challenging, as the contention among the cores for various shared resources leads to inherent timing unpredictability. This paper proposes the use of shared cache in a predictable manner through a combination of locking and partitioning mechanisms. We explore possible design choices and evaluate their effects on the worst-case application performance. Our study reveals certain design principles that strongly dictate the performance of a predictable memory hierarchy.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Measurement, Performance

Keywords

Shared-cache multi-core, WCET, cache locking, cache partitioning

1. INTRODUCTION

Multi-core architectures are increasingly common in both desktop and embedded markets. Energy and thermal constraints are effectively precluding the design of complex high-performance single-core processors. In this context, multiple simpler processing cores on a single chip is an attractive option. Several manufacturers (e.g., Intel, AMD) have released dual/quad cores while Sun's Niagara multiprocessor accommodates 8 cores on the same die. In the embedded domain, ARM MPCore is a synthesizable multi-processor configurable to contain between 1 and 4 ARM11 cores. IBM Cell processor in Sony PlayStation 3 contains 9 cores while Xenon in Microsoft's Xbox 360 is a custom PowerPC-based triple-core.

Memory optimizations remain indispensable for performance as we move from single-core to multi-core systems. Along with the processing cores, increasingly large memory area can now be integrated onto the die. A popular choice for the on-chip memory structure is a two-level cache hierarchy (e.g., Niagara, Xenon, Power5). In this architecture, Level 1 (L1) caches are attached to and privately accessible by each core. All the cores share the access to a large Level 2 (L2) cache. The presence of a shared cache offers the flexibility in adjusting the memory allocated per core according to its requirement, as well as the possibility of multiple cores enjoying fast access to shared code/data. This potentially improves the performance, but also requires complex resource management.

The challenge is even greater when designing for real-time applications, which cannot afford to miss deadlines and hence demand timing predictability. Any schedulability analysis requires the worst-case execution time (WCET) of the application's tasks as input. Static analysis of a task to estimate its WCET bound is complicated by the dynamic nature of cache memory. Still, a decade of research has solved the problem of cache-aware WCET analysis to a large extent in the context of single-core processors. However, the interaction and the resulting contention among multiple cores in a shared cache bring out many new challenges. To the best of our knowledge, no static analysis method has been developed to estimate WCET bounds in the presence of shared caches. In our opinion, it will be extremely difficult, if not impossible, to develop such a method that can accurately capture the contention. Instead, we propose to use the shared cache in a restrictive manner that eases the analysis effort, with possible tradeoff in performance. Towards this end, we exploit two mechanisms: *cache locking* and *cache partitioning*. Cache locking allows the user to load selected contents into the cache and subsequently prevents these contents from being replaced at runtime. This feature is available in several commercial processors (PowerPC 440 core, ARM 920T, Freescale Semiconductor's e300 core, etc). Cache partitioning assigns a portion of the cache to each task (or processor), and restricts cache replacement to each individual partition. Cache partitioning enables compositional analysis where the timing effect of each task (processor) can be estimated separately.

The interplay among processing elements in a multi-core setting through the shared cache provides some unique design choices and opportunities. One simple choice, for example, is global cache locking where contents are selected from all the tasks in all the cores. More sophisticated policies may partition the cache among the cores or the tasks and manage each partition independently. The relative merits of these different design choices are not obvious.

In this paper, we explore the possible design choices for a predictable and high performance shared L2 cache on multi-core architectures. We devise different combinations of cache locking and partitioning schemes, then study their impact on the worst-case performance of applications with different characteristics. This

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

study can provide guidelines to real-time application programmers in terms of design decisions for the memory hierarchy.

2. SHARED CACHE MANAGEMENT

We consider a multi-core architecture consisting of identical cores. The on-chip memory is configured as two-level caches with a *shared L2 cache*, which is the focus of this work. We assume that the cache coherence is implemented in hardware, and that the caches support locking and set-based partitioning [4] that allocates a number of sets (rows) to each task. This paper focuses on *instruction cache*, though our technique is equally applicable to data caches.

We adopt the classic real-time system model where a set of independent tasks $\{T_1, T_2, \ldots, T_K\}$ execute periodically. Each task T_i is associated with a period p_i, which defines its deadline, and a worst-case execution time c_i. We choose *partitioning* [2] strategy for homogeneous multiprocessor scheduling. In a partitioning strategy, once a task is allocated to a processor, it is executed exclusively on that processor. Any uniprocessor scheduling algorithm can then be applied on each processor. Partitioning strategy has the advantage of lower overhead compared to global strategy that allows migration of a task to a different processor at runtime.

López et al. [6] show that the earliest deadline first (EDF) scheduling policy with First Fit (FF) allocation is an optimal partitioning approach with respect to utilization bounds. Our framework applies this policy. FF assigns a task to the first processor that can accept it. A task set is EDF-schedulable on uniprocessor if $U \leq 1$, where U is the utilization of a task set $\{T_1, T_2, \ldots, T_K\}$ given by $U = \sum_{i=1}^{K} \frac{c_i}{p_i}$. The *system utilization* of a Q-core multiprocessor is $U_{system} = \frac{U}{Q}$. We measure the performance of a task set on a multiprocessor by the system utilization: the lower, the better.

We separate the treatment of the private L1 caches and the shared L2 cache, in order to observe the shared cache behavior while abstracting out the effects of the L1 caches. As our focus is on the shared cache, we choose a simple static locking scheme for L1. The private L1 cache attached to a core is utilized only by the tasks executing on that core; for each, we adopt the cache content selection algorithm for multitasking systems [9]. The chosen blocks for L1 will be excluded during content selection for the L2 cache.

The shared L2 cache opens up the opportunity to combine different locking and partitioning schemes as shown in Figure 1(e). For cache locking, we can choose a *static* scheme (cache content remains unchanged throughout execution) or a *dynamic* scheme (cache content can be reloaded at runtime) For cache partitioning, we have the choice of (1) *no partition*, where a cache block may be occupied by any task, scheduled on any core; (2) *task-based partition*, where each task is assigned a portion of the cache; or (3) *core-based partition*, where each core is assigned a portion of the cache, and each task scheduled on that core may occupy the whole portion while it is executing. From these, the {*dynamic locking, no partition*} combination must be ruled out, because dynamic locking strictly requires a dedicated partition. Further, both the {*static locking, no partition*} (SN) and the {*static locking, task-based partition*} (ST) schemes lock the cache contents chosen from all tasks in the application throughout execution, but SN offers more flexibility by not enforcing a concrete boundary. *Thus ST is either inferior or at most as good as SN; we eliminate ST from our evaluation.*

Figure 1(a–d) illustrates the four eligible possibilities, applied on a multi-core with 2 processing elements (PE_1, PE_2) and 4 independent tasks ($T_1, \ldots T_4$). The scheduler assigns T_1, T_2 to PE_1 and T_3, T_4 to PE_2. T_1 and T_4 are each divided into two regions for dynamic cache locking. We assume a 2-way set-associative shared L2 cache with 8 sets.

Static Locking, No Partition (SN). This is the simplest scheme where the cache content is kept unchanged throughout application runtime (Figure 1(a)). A cache block can be assigned to any task irrespective of the processor it is scheduled on. This scheme offers maximum flexibility; however, its performance is restricted if the code size of all the tasks together far exceeds the L2 cache size. For static locking, we apply the cache content selection algorithm presented in [9], which minimizes the system utilization.

Static Locking, Core-based Partition (SC). On a system with preemptive scheduling, only memory blocks belonging to the "active" tasks are useful at any time. When a task T_i is preempted by $T_{i'}$ on one of the cores, we can replace T_i's memory blocks in the cache with those of $T_{i'}$'s. This comes at the cost of reloading the cache at every preemption. This scheme requires the cache to be partitioned among the cores, as each core has an active task at any point of time and the cores invoke preemptions independently. However, when a task runs on a core, it can occupy the entire partition for that core. In Figure 1(b), PE_1 gets the first four sets; PE_2 gets the rest. Initially T_1 occupies PE_1's partition and T_4 occupies PE_2's partition. When T_2 preempts T_1, it loads and locks PE_1's partition with its own content. We adapt a dynamic programming based optimal partitioning algorithm [11] here.

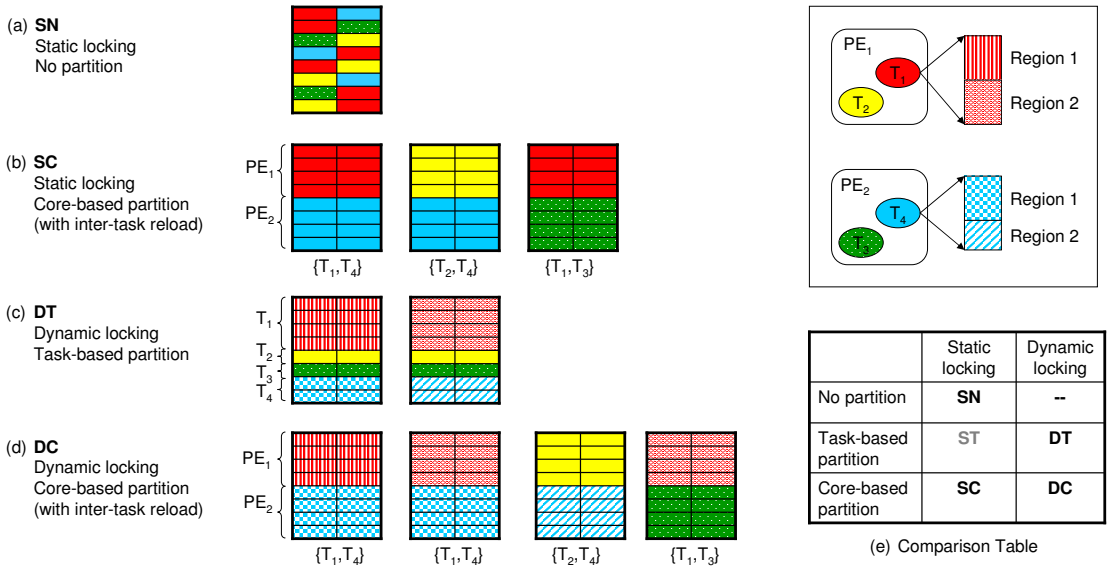
Dynamic Locking, Task-based Partition (DT). Dynamic locking allows more memory blocks to fit in the cache via runtime load and lock. The overhead is the cache reload cost every time the execution of a task moves from one region to another. As different tasks have different region formations, the cache is first partitioned among the tasks. Each task then performs dynamic locking within its partition. Figure 1(c) shows the scheme at work for T_1 and T_4. In contrast to SC, reloading is performed intra-task. No inter-task reloading is required as the partitioning prevents interference among the tasks, thus preemptions incur no cache reload overhead. However, if the application comprises a large number of tasks, such rigid partitioning might not be effective. DT also suffers from the same drawback as SN: tasks occupy cache blocks even when they are inactive (preempted). We employ the dynamic locking algorithm in [8] here.

Dynamic Locking, Core-based Partition (DC). In this most complex scheme, reloading is supported within a task in addition to reloading at preemption (see Figure 1(d)). Initially, the cache is loaded with region 1 of T_1 and region 1 of T_4. As time progresses, T_1's execution (on PE_1) moves to region 2, which then replaces the content of PE_1's portion of the cache. Later on, a preemption brings into the cache the content associated with T_2. However, when T_1 resumes, it again brings in region 2 into the cache.

3. EXPERIMENTAL EVALUATION

We choose a dual-core multiprocessor for experimental evaluation. Our pool of tasks comprises independent C programs chosen from popular WCET benchmarks. From this pool, we construct task sets of 4 tasks each. These task sets are chosen to exhibit different characteristics (region formation, code size, and WCET) that affect the effectiveness of the locking and partitioning schemes, as listed under each chart in Figure 2. Task periods are set such that the utilization of each core is very close to 1, that is, the task set is schedulable with maximum system utilization.

For both L1 and L2 caches, we choose 16 byte line size (2 instructions) and 2-way associativity. The size of L1 caches are fixed at 128 byte per core. The small size is chosen so that memory requirements are mostly served from the shared cache, enabling better observation of the multiprocessing effect. Given the average



	Static locking	Dynamic locking
No partition	**SN**	**--**
Task-based partition	**ST**	**DT**
Core-based partition	**SC**	**DC**

Figure 1: Different locking and partitioning schemes for the shared L2 cache

code size of 4 KB per task, we vary the size of the shared L2 cache from 1 KB to 4 KB. We assume L1 latency is 1 clock cycle, L2 latency is 4 cycles, and off-chip latency is 10 cycles.

For all task sets, we first select the L1 contents for each core, then apply the four shared caching schemes *SN, DT, SC* and *DC*. This gives us the set of locked memory blocks for each task. Note that a task may not be allocated any space in the cache in certain schemes. Based on these locked sets, we calculate the new WCET of each task, taking into account any extra latency involved to reload the cache contents. Finally, we compute the system utilization. Figure 2 shows the system utilization computed for each task set given the various schemes and varied cache sizes. These values are normalized against the system utilization without a cache.

Cache Partitioning Strategy. We have three possible choices for cache partitioning as shown in Figure 1(e): *no partition, task-based partition,* and *core-based partition*. The experimental results indicate that the best partitioning choice strongly depends on the locking scheme (static or dynamic) used in conjunction and cache size.

First let us consider static cache locking. As mentioned before, task-based partition (*ST*) is provably inferior compared to no partition (*SN*). This is due to the partitioning granularity of *ST*, which requires the cache partition for a task to contain power of 2 sets. *SN*, in contrast, allows complete flexibility and a task is free to occupy as little as one cache block.

However, when it comes to no partition versus core-based partition (*SN* versus *SC*), the situation is reversed; *SC* consistently performs better than *SN* irrespective of cache size and application characteristics. Recall that in *SC*, the entire partition for a core is occupied only by the "active" tasks at any point of time. For *SN*, in contrast, the "idle" tasks continue to occupy precious real-estate in the shared cache. The downside of *SC* is of course the cache partition reloading and locking cost at every preemption. However, the results indicate that preemption cost does not over-shadow the advantage of better cache utilization by *SC*.

The best partitioning strategy, when used in conjunction with dynamic cache locking scheme, depends heavily on cache size. Here we are comparing *DT* versus *DC*). As dynamic locking allows better cache utilization through intra-task reloading at region boundaries, *DT* becomes a competitive scheme. Its main advantage is zero interference among the tasks, thus avoiding cache reloading at task preemption. *DC*, on the other hand, offers more cache space per task, thus performs better at small cache sizes. As cache size increases, the difference between the two is negligible.

Static versus Dynamic Cache Locking. Here the best choice is strongly influenced by application characteristics, cache size, and the cache partitioning strategy.

Let us first consider core-based partition (*SC* versus *DC*). Clearly, *DC* can better utilize the cache if the application has many hot regions and the cache space is limited. The experiments validate this observation with *DC* performing better for task sets with large number of regions (C*RLT*). Even for those task sets, *DC* gives diminishing returns as cache size increases. If the constituent tasks do not contain profitable regions (C*RST*), then it is better to use *SC*, which is a much simpler cache management scheme.

We now turn to task-based partition. Normally, we would compare *ST* against *DT*, but as already discussed, *ST* is never better than *SN*. We thus compare *DT* with *SN* instead. The trend is roughly the same as *SC* versus *DC*: *DT* wins if the tasks have a number of hot regions. However, there is one important difference. As we choose a reload point within a task only if it is profitable to do so, *DC* cannot perform worse than *SC*. We cannot make this claim for *DT* over *SN*, as *DT* and *SN* use different schemes for cache partitioning. Indeed, *SN* enjoys much more flexibility in cache allocation to tasks. The rigid partitioning of *DT* can completely over-shadow the gain from reloading at region boundaries. Thus *SN* can perform better than *DT* when (1) tasks have very few regions (CSRSTL and CSRSTS) or (2) some tasks get no space in the shared cache (CSRLTL) with *DT* policy.

Guiding Design Principles. From the preceding discussion, we can make the following general conclusions.

- *SC* is better than *SN*, which in turn is better than *ST* irrespective of cache size and application characteristics. In other words, if the designer wants to use static cache locking, she should use core-based partition.

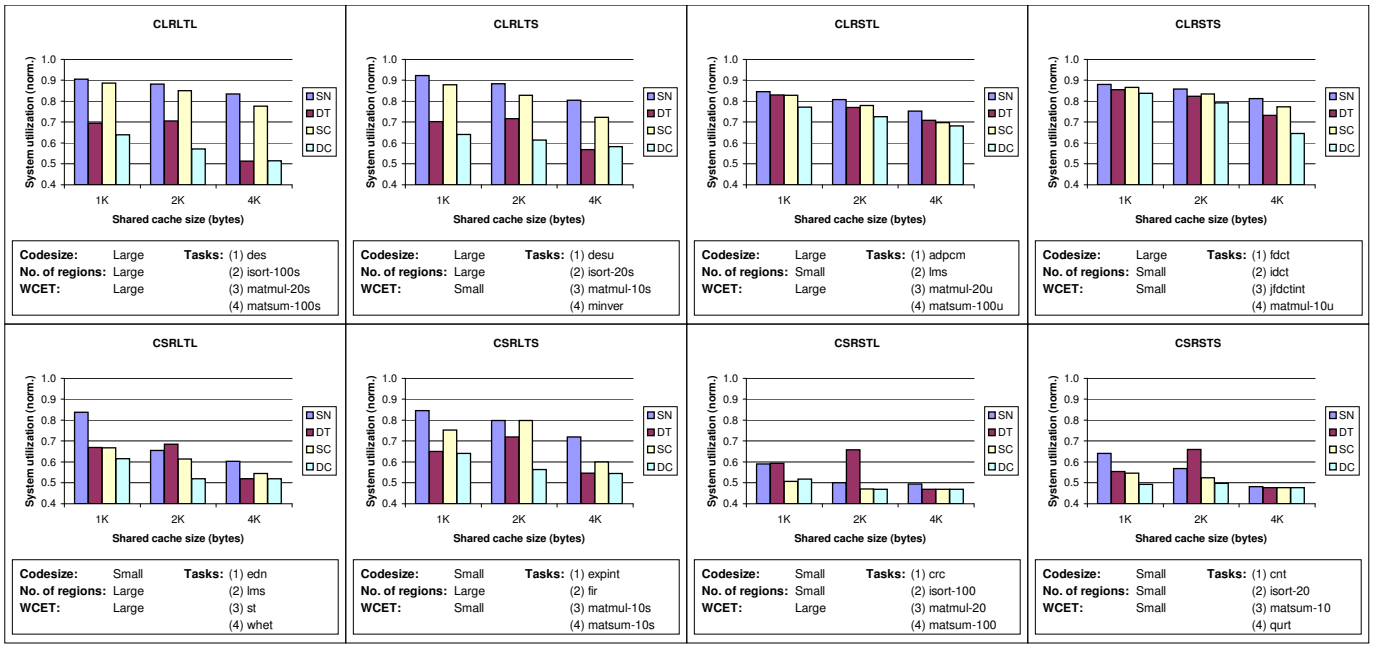


Figure 2: Effects of shared caching schemes SN, DT, SC, and DC on task sets with various characteristics

- *DC* is better than *DT* for small shared cache size. They are comparable for large cache size; avoiding cache reloading at preemption via task-based partitioning does not seem to affect performance much. *Core-based cache partition emerges as overall winner independent of locking strategy*.
- Dynamic cache locking is better than static cache locking *only* for tasks with a large number of hot regions and for smaller shared cache size. Moreover, designer should be extra careful to use task-based partition in conjunction with dynamic cache locking. If some tasks do not get any cache allocation, the overall system utilization is severely affected.

4. RELATED WORK

The state of the art in the domain of predictable hard real-time systems shows a lot of research effort in modeling dynamic cache behavior for WCET estimation [5, 7, 13]. Static cache locking algorithms to minimize system utilization are presented in [9], while a dynamic cache locking algorithm is proposed in [8]. In the context of data caches, [14] tries to balance the performance and predictability tradeoff introduced due to locking, by applying it only on parts of the program that are difficult to analyze statically. All these methods work on private caches. Hardware-based cache partitioning schemes have been presented in the literature; [4] allows set partitioning while [3] proposes way-partitioning. The technique in [11] attempts to minimize task utilization via dynamic programming, while [12] aims to minimize cache miss rate. The partitioning technique in [10] allows prioritizing of critical tasks. Finally, for shared caches, [1] proposes a cache-aware multi-core scheduling scheme for real-time applications.

5. CONCLUSION

In this paper, we have explored predictable caching schemes for shared memory multi-cores in the context of preemptive hard real-time systems. In particular, we have developed and evaluated various design choices for the shared L2 cache by exploiting static/dynamic locking and task/core-based partitioning. We have studied system utilization for the different choices with respect to the characteristics of the task set and cache size. Our study reveals some interesting guiding principles for real-time system designers with respect to the memory hierarchy.

6. ACKNOWLEDGEMENTS

This work was partially supported by NUS project R-252-000-292-112 and A*STAR SERC project R-252-000-258-305.

7. REFERENCES

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *RTAS*, 2006.
[2] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004.
[3] D. Chiou. *Extending the reach of microprocessors: column and curious caching*. PhD thesis, MIT, 1999.
[4] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *RTSS*, 1989.
[5] Y.-T. Li and S. Malik. *Performance Analysis of Real-time Embedded Software*. Springer, 1998.
[6] J. López, M. García, J. Díaz, and D. García. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *ECRTS*, 2000.
[7] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3), 1999.
[8] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS*, 2006.
[9] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.
[10] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *CASES*, 2007.
[11] J. Sasinowski and J. Strosnider. A dynamic programming algorithm for cache memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8), 1993.
[12] G. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *PDCS*, 2001.
[13] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.
[14] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *SIGMETRICS*, 2003.