Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses

[1]Noriaki Suzuki, [2]Hyoseung Kim, [2]Dionisio de Niz, [2]Bjorn Andersson, [2]Lutz Wrage, [2]Mark Klein, and [2]Ragunathan (Raj) Rajkumar

n-suzuki@ha.jp.nec.com, hyoseung@cmu.edu, {dionisio, bandersson, lwrage, mk}@sei.cmu.edu, raj@ece.cmu.edu

[1]NEC Corporation, Japan [2]Carnegie Mellon University, USA

***Abstract*—In commercial-off-the-shelf (COTS) multi-core systems, the execution times of tasks become hard to predict because of contention on shared resources in the memory hierarchy. In particular, a task running in one processor core can delay the execution of another task running in another processor core. This is due to the fact that tasks can access data in the same cache set shared among processor cores or in the same memory bank in the DRAM memory (or both). Such cache and bank interference effects have motivated the need to create isolation mechanisms for resources accessed by more than one task. One popular isolation mechanism is *cache coloring* that divides the cache into multiple partitions. With cache coloring, each task can be assigned exclusive cache partitions, thereby preventing cache interference from other tasks. Similarly, *bank coloring* allows assigning exclusive bank partitions to tasks. While cache coloring and some bank coloring mechanisms have been studied separately, interactions between the two schemes have not been studied. Specifically, while memory accesses to two different bank colors do not interfere with each other at the bank level, they may interact at the cache level. Similarly, two different cache colors avoid cache interference but may not prevent bank interference. Therefore it is necessary to *coordinate* cache and bank coloring approaches. In this paper, we present a coordinated cache and bank coloring scheme that is designed to prevent cache and bank interference simultaneously. We also developed color allocation algorithms for configuring a virtual memory system to support our scheme which has been implemented in the Linux kernel. In our experiments, we observed that the execution time can increase by 60% due to inter-task interference when we use only cache coloring. Our coordinated approach can reduce this figure down to 12% (an 80% reduction).**

I. Introduction

In multi-core systems, the execution of one task on one processor core can depend on the execution of a task on another processor core which is a major concern for hard real-time systems. This dependency is mostly caused by the following effects:

- E1. Eviction of cache blocks in a cache memory shared between processor cores.
- E2. Contention on the bus between the memory controller and the DRAM modules.
- E3. Eviction of the currently open row in the memory bank in DRAM modules.
- E4. Reordering of memory requests in the queue of the memory controller, caused by the memory controller favoring memory access to the currently open row in a memory bank.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0000537

	Works with COTS Multicore	Deals with effect			
		E1	E2	E3	E4
Analysis method					
[16]	No	No	Yes	No	No
Mechanism					
[15]	No	No	Yes	No	No
[16]	No	No	Yes	No	No
[14]	No	No	Yes	Yes	Yes
[5]	No	Yes	Yes	Yes	Yes
Analysis method					
[4]	Yes	No	Yes	No	No
[10]	Yes	No	Yes	No	No
[13]	Yes	No	Yes	No	No
Mechanism					
[9]	Yes	No	No	Yes	Yes
[11]	Yes	Yes	No	No	No
[17]	Yes	No	Yes	No	No
Our work	Yes	Yes	No	Yes	Yes

TABLE I: Comparison with previous work.

Consequently, the research community has developed (i) methods for analyzing the impact of these dependencies and (ii) run-time mechanisms that protect the execution time of one task from these effects. Table 1 summarizes the state-of-art.

We have heard from software practitioners a strong preference for COTS multicore systems and therefore we center our discussion on those. The queuing discipline used for resolving multiple requests to use the memory bus is often undocumented in COTS multicore systems and consequently, knowing exactly which request will be served at any given time is difficult. Therefore, E2 has been dealt with [4], [13] by developing analysis techniques assuming that the arbitration for the memory bus is work conserving but making no other assumption on the arbitration. E2 has also been dealt with [17] by developing mechanisms where a task is assigned a certain number of bus accesses that it is allowed to generate and, at run-time, periodically monitoring (using performance monitoring counters) the number of cache misses a task generates and if the number of cache misses reaches a high value (close to the allowed number) the task is suspended (because it has generated a large number of memory transfers on the bus).

Cache coloring is an often-favored [11] technique for providing protection against E1. This technique sets up the virtual-to-physical-address translation so that no two tasks access the same cache set in the shared cache and hence one task cannot evict a cache block that another task has fetched to the shared cache. The same idea has been used for dealing with E3 and E4, and is referred to as *bank coloring* [9]. Software practitioners clearly benefit from using both cache coloring and bank coloring but while the virtual address translation is used for both of them, it is not clear how to configure the translation

to achieve both at the same time, and the research literature provides no guidance on how to achieve such coordinated bank and cache coloring.

Therefore, in this paper, we present coordinated bank and cache coloring. Specifically, we make the following contributions:

- We create the *first* cache-bank color conflict model that describes the conflicts that a cache coloring scheme has with bank coloring schemes in generic commercial multi-core processors.
- We develop the *first* operating systems (OS) mechanism that provides *coordinated* cache and bank coloring.
- We create the *first* algorithm that *coordinates* the allocation of tasks to cores and bank and cache colors to tasks.

The remainder of the paper is organized as follows. Section 2 gives a background on memory hierarchies. Section 3 gives an introduction to cache and bank coloring. Section 4 presents algorithms for allocating processor cores, cache colors, and bank colors to tasks. Section 5 gives conclusions.

II. Bank and Cache Interference

In this section we discuss the nature of the task interference related to cache and memory banks. We start by providing a general introduction to the memory hierarchy to set the context. Then we discuss cache and bank interference separately followed by the interactions between them.

A. Memory Hierarchy Background

To cope with the stark difference between the processor and the memory access speeds, today's processors implement a memory hierarchy that combines memory elements of different speeds. These elements, ordered in decreasing access speed and increasing memory capacity, start with processor registers, followed by multiple levels of cache, and end with memory banks. As a stream of instructions executes in the processor, its data (and the instructions themselves) is loaded first in the last-level cache then in each of the cache levels closer to the processor and finally in the processor registers. Repeated uses of the same data are sped up by accessing them in the fastest memory where it is loaded. Given that the set of data accessed by a program can exceed the size of the memory element used, old data is evicted and replaced by the newly accessed data. This eviction induces a longer access time the next time this data is accessed given that it will need to be accessed from a slower memory element (down to the main memory). While some evictions are triggered by the same task that loaded the data, it can also be caused by another task running in the same or other cores in the processor.

In this paper we study the evictions across tasks in shared cache and memory banks. However, the nature of these two types of evictions and the interference they induce are different. In the following we explain these two types in detail.

1) Cache Eviction and Interference: The cache hierarchy takes advantage of spatial and temporal locality of memory accesses to preserve the data that is more likely to be accessed in the fastest cache possible. Specifically, it is well known that when a program accesses a memory location, it is highly likely for the next few accesses to be close the same location. This location closeness is known as *spatial locality* for which caches are organized in cache lines of multiple bytes (typically 64) that are loaded together. That is, when a memory location is accessed and loaded into cache, the surrounding memory locations in the same cache line are loaded as well. In this way, the next location accessed by the program will likely be already loaded in the cache line and its access will be much faster.

Temporal locality, on the other hand, relates to the fact that memory locations accessed in the past are likely to be accessed in the future again. Given the lack of an oracle that can tell us the future, the cache mechanisms assume that it is more likely that a recently-accessed location will be accessed again instead of a less recently-accessed one. This assumption is implemented in cache-replacement policies that select some approximation of a least-recently-used cache line to be evicted when no empty cache line is available, since it would be the least likely to be accessed again.

When two (or more) tasks access the same cache in an interleaved fashion they break each other's temporal and spatial locality inducing costly evictions. When both tasks are running on the same core, these interleavings are the result of the context switch between tasks and happens only when the scheduler decides to perform a context switch. The effect of this eviction, known as *Cache-Related-Preemption Delay* (CRPD), has been studied in the past but present a significant schedulability penalty with limited predictability. When the tasks are in different cores, the interleaving is continuous as they run in real parallelism and can produce evictions perhaps more costly and less predictable.

The cache is partitioned into sets restricted to contain words from different regions (memory ranges) of main memory. This set partitioning prevents memory locations restricted to be loaded in one cache set from evicting data loaded from memory addresses that are loaded in another cache set. We will use this property to implement our partitioning scheme to be discussed shortly.

2) Memory Bank Evictions and Interference: Main memory is divided into memory banks with the main purpose of parallelizing the access to the physical devices and reducing the speed difference with the processor speed. These banks are organized into ranks, that bundle multiple banks together, and ranks are grouped into channels. Different banks, ranks, and channels can be accessed in parallel with very little interference within each other.

Memory banks are internally organized in rows and columns. A row is a sequence of memory locations that are divided into columns. A column within a row is the minimum amount of memory transfer from/to the banks. To access a memory address the memory controller first identifies the corresponding channel, rank, bank, row, and column where the memory address is located. Then, it transfers the row where the address is located to the bank row buffer. Finally, it accesses the column where the address is located. Subsequent accesses to columns from the same row can be performed directly from

the row buffer saving the time to load such a buffer. However, if an address mapped to a different row is accessed then the row buffer needs to be reloaded and the previous data is evicted.

Row buffers implement a caching strategy similar to the cache line, matching the spatial locality of the program. The temporal locality in memory banks, however, have a different flavor from the locality of the cache. That is, while in the cache memory the temporal locality is implemented as a replacement policy (e.g. least-recently used) that predicts the future based on the past, to access locations in memory banks the memory controller has more information. In particular, memory access requests to main memory arrive at the memory controller from multiple cores at the same time. Given that the access to the memory banks is much slower than the cache, the memory controller keeps a queue of requests waiting to be served. This queue contains in fact the sequence of future accesses to the banks, and hence, the memory controller does not need to guess this future. More importantly, the memory controller is able to change the future through the reordering of the requests in this queue. Specifically, memory controllers reorder the request queue to avoid accesses to other rows to get in the middle of a sequence of request to the same row. This reordering reduces the row buffer evictions and improve the throughput of the whole system and the program whose accesses were favored in the reordering. Unfortunately, the reordering can also enlarge the worst-case execution time of the tasks. However, as happens with different cache lines, accesses to different banks do not interfere with each other. Furthermore, it is worth noting that the reordering effect is only significant between accesses from different cores given that the effect that happens in the context switch between two tasks running on the same core is negligible.

III. Coordinated Cache and Bank Memory Coloring

The mapping of a memory location to specific cache sets or memory banks is determined by its address. For cache sets, a subset of the address bits are used as the cache set *index*. This index uniquely identifies the cache set where this address can be loaded. This means that if we restrict different tasks to use memory addresses with different cache index then we avoid cache interference between them.

Memory banks, on the other hand, are selected with a subset of address bits that identify the channel, rank, and bank numbers. In the same fashion as cache, restricting different tasks to use different bank numbers eliminates bank interference.

Both bank and cache coloring are implemented by allocating physical pages with different cache-set indexes (or bank numbers) to different tasks and hence it is also known as page coloring.

A. Cache-Color Address Bits

In order to implement cache coloring it is necessary to first identify the address bits that identify the cache index. In order to do this it is possible to obtain the cache parameters from processors specification. In particular, it is necessary to obtain: (1) the cache size, (2) the cache line size, and (3) the set associativity. With these data it is possible to calculate the number of address bits needed to identify the cache set (C) of an address with the formula $C = \log_2\left(\frac{S}{WL}\right)$ where S is the cache size, W the number of ways of the cache, and L the cache line size.

Last-Level Cache Size (S)	8 MB
Number of Ways (W)	16 × 4
Cache Line Size (L)	64 bytes

TABLE II: Intel Core i7-2600 Processor Cache Specification

Rows	Rank	Bank	Col 1	Channel	Col 2	Byte
31-18	17	16-14	13-7	6	5-3	2-0

Fig. 1: Typical Bank Address Layout

To locate the starting bit of the cache index address bit we use the size of the cache line (L) and apply log base 2 ($log_2(L)$). As an example consider the specifications of the Intel Core i7-2600 processor presented in Table II.

In this case the number of cache sets is calculated as $C = log_2\left(\frac{8MB}{64 \cdot 64}\right) = 11$.[1] Then the starting bit can be located at $log_2(64) = 6$ extending up to bit 16.

While all the cache index address bits could potentially be used to create different colors, the virtual memory system only allows us to control the address bits that are included in the page frame number. In other words, a memory address is divided into a page number and an byte offset inside the page. The virtual memory system works by replacing virtual memory page number with physical page numbers, a.k.a. frame numbers. As a result, only the address bits that are included in the identification of the frame number are available to the virtual memory manager. This depends on the page size that is typically configured as 4KB by the OS. When the page size is 4KB, the address bits for the page frame number start from bit 12. As a result, cache coloring can control only 5 bits that range from bit 12 to bit 16.

B. Bank-Color Address Bits

The address bits used to indicate individual memory banks can be found in a similar fashion to cache bits. Figure 1 presents a typical layout of the address bits for banks [6]. There are three features in this layout that are worth noting. First, the channel is located in the low-order bits (6). This is aimed at interleaving the memory access across channels to allow parallel access to consecutive cache lines in a pipelined fashion. Secondly, the bank bits are in lower-order bits than the rows. This has the same motivation of the channel favoring the interleaving of access to banks to parallelize memory accesses across banks for the same row. Finally, the rows are the most significant bits in the layout. This has the intention to minimize the likelihood of changing rows when accessing consecutive memory. All of these features are aimed at improving the access time of the memory for a single task. Unfortunately, they do not help prevent the interference across tasks, which is our goal.

In modern processors the bank address bit layout in Figure 1 is augmented with a randomization strategy to minimize

[1]The shared cache of the Intel Core i7 processor consists of four cache slices, which makes the number of ways of each cache slice be multiplied by four. More details on this can be found in [7].

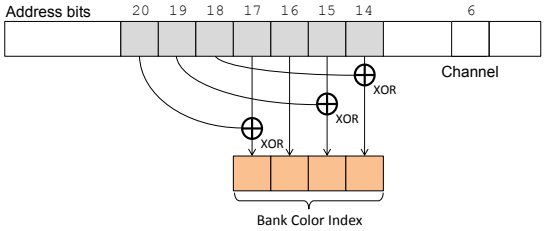

Fig. 2: Randomization Bank Bit Strategy (XOR) of Intel Core i7-2600

3 Bit Value	Bank Color	Cache Color
000	00	00
001	00	01
010	01	10
011	01	11
100	10	00
101	10	01
110	11	10
111	11	11

TABLE III: Bank and Cache Colors Intersection Example

the likelihood of bank collisions. This strategy is implemented by XORing row bits with the bank bits to produce the final bank bits. For instance, Figure 2 presents the xor strategy of the Intel Core i7-2600 memory controller for the four single-sided DIMMs of 2GB each.

It is worth noting that modern processors do not publish the bank address bits. As a result, previous work has been published to discover this address bits in a experimental fashion [9]. We took this work and created an improved procedure (not presented due to space constraints).

As happens with cache coloring, bank coloring can only take advantage of the bits that can be controlled by the virtual memory manager which in our case rules out the channel bit (number 6).

C. Bank and Cache Color Interaction Model

A page color is a collection of pages that do not interfere with each other. When pages are colored to avoid cache interference they guarantee that pages of different colors will not evict cache lines from each other. Similarly when they are designed based on memory banks they guarantee that they do not map to the same bank and hence do not induce row buffer eviction or reorder memory requests. In this paper we aim at avoiding both cache and bank interference and, hence, we require a coloring scheme that avoids both types of interference. Unfortunately, not only do cache and bank colors intersect but neither subsumes the other. This means that it is not possible to build a color hierarchy where one type or color is subsumed into another, e.g., cache colors within bank colors.

The bank and cache color intersection is caused by the intersection of the address bits that identify these colors. To illustrate this intersection consider a memory system where the color of both the cache and the banks are identified by two address bits and one of the bits is between the two. Specifically bit i and $i+1$ identify bank colors while $i+1$ and $i+2$ identify the cache colors. Both cache and bank colors can be represented by all possible values of these three bits, leading to the intersection of colors depicted in Table III.

Cache \ Bank	00	01	10	11
00	X		X	
01	X		X	
10		X		X
11		X		X

Fig. 3: Bank and Cache Color Intersection Matrix

Note that the intersection is only partial. For instance, the bank color 00 in our example intersects with cache colors 00 and 01 but not with 10 or 11. Similarly, cache color 00 only intersects with bank colors 00 and 10.

In order to better understand the impact of the color intersection we use an intersection matrix where columns represent bank colors and rows cache colors. Figure 3 shows the matrix for the color scheme of Table III with an *X* marking the colors that intersect.

The matrix from Figure 3 simplifies the visualization of both the color intersections and the intersection gaps. This allows us to highlight two additional facts. First, the collection of all cells with an *X* in the intersection matrix represents the whole memory space[2]. Secondly, each cell with an *X* represents a subset of pages that do not intersect with other subsets of pages in other cells, either through cache or memory banks.

The intersection of bank and cache bits reduces the number of independent cells in the intersection matrix. This can be reflected as a reduction of the number of cache colors available whenever a bank color is selected. Specifically, the number of bank colors is $B = 2^{\text{numbankbits}}$, and the number of cache colors per bank is $H = 2^{(\text{numcachebits}-\text{numintersectingbits})}$.

1) Bank Address Bit Randomization: The bank-bits randomization scheme discussed in Section III-B has the effect of removing the gaps in the intersection matrix. This is because it is possible to set the common bits to whatever value is needed to produce a particular cache bit and adjust the corresponding XORed bits accordingly. Figure 4 presents the variation of the intersection matrix from Figure 3 when XORing mechanism is added. In this case, for a given two-bit value of the cache color we can select a two-bit bank value that have the same shared bit value along with the corresponding row bits that produces the desired bank color. For instance, if we select the cache bits 10 then, to obtain the bank color 00 (non-existing in Figure 3), we can use bank bits 11 along with row bits 11 to produce bank color $11 \oplus 11 = 00$. All row and bank bit combinations are shown in the bank-color heading of Figure 4. For this case, if we calculate the number of bank colors as $B = 2^{\text{numbankbits}}$, then the number of cache colors per bank are calculated as $H = 2^{\text{numcachebits}}$. Note that for this case, *numintersectingbits* does not play a role in computing H.

D. Memory Interference Model

We now define our new memory interference model. We define a system as

$$\mathcal{S} = (\tau = \{\tau_1, \ldots, \tau_n\}, \pi = \{\pi_1, \ldots, \pi_m\}, B, H)$$

where

- τ_i is a task defined as $\tau_i = (T_i, \{C_i^1, \ldots, C_i^k\}, M_i)$ with a period T_i, a set of execution times where

[2]In the rest of this paper we refer to these cells as the memory cells and we will use them to discuss our memory interference model and allocation algorithms.

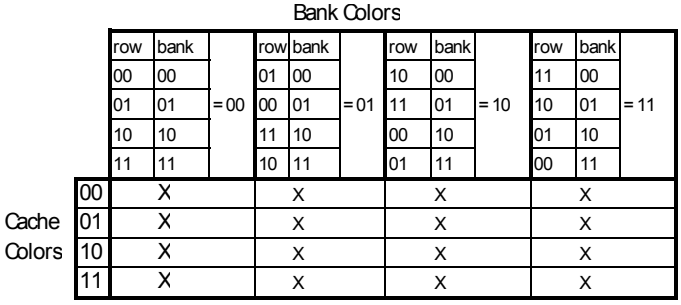

Fig. 4: Intersection Matrix with XOR.

each C_i^j represents the worst-case computation time without interference when task τ_i is assigned j cache colors, and a memory requirement M_i that represents the number of memory cells as shown in Figure 4.

- π_j represents a processor core.
- B is the number of bank colors.
- H is the number of cache colors.

We will define a concrete allocation and an abstract allocation where the former states which specific set of colors are assigned to a task and the latter specifies only the number of colors assigned. A concrete allocation is defined as

$$\mathcal{CA} = \{ca_i = (P_i, \mathbb{B}_i, \mathbb{H}_i)\}$$

where

- $P_i \in [1, m]$ identifies the processor where τ_i is allocated. For convenience we use the value 0 (zero) when a task is not assigned to any processor.
- $\mathbb{B}_i$ is the set of bank colors assigned to task τ_i.
- $\mathbb{H}_i$ is the set of cache colors assigned to task τ_i.

An abstract allocation is defined as

$$\mathcal{AA} = \{aa_i = (P_i, B_i, H_i)\}$$

where

- $P_i \in [1, m]$ identifies the processor where τ_i is allocated.
- B_i is the number of bank colors assigned to task τ_i.
- H_i is the number of cache colors assigned to task τ_i.

We will now define memory interference. For this purpose, we introduce two predicates below.

Definition 1: conflict($\mathcal{CA}$) = $\exists ca_i, ca_j \in \mathcal{CA} | (i \neq j) \wedge ((\mathbb{B}_i \cap \mathbb{B}_j \neq \emptyset \wedge P_i \neq P_j) \vee (\mathbb{H}_i \cap \mathbb{H}_j \neq \emptyset))$

This encodes that a conflict exists if a bank color is shared across cores or a cache color is shared across tasks.

Definition 2: conflict($\mathcal{AA}$) = $((\sum_{\pi_j \in \pi} (\max_{i \in [1,n] | P_i = j} B_i)) > B) \vee ((\sum_{i \in [1,n]} H_i) > H)$

In this case a conflict exists if the sum of the number of bank colors assigned to each core exceeds the number of available bank colors or the number of allocated cache colors exceeds the available colors. The bank colors assigned to a core can be shared among all tasks assigned to this core without causing interference.

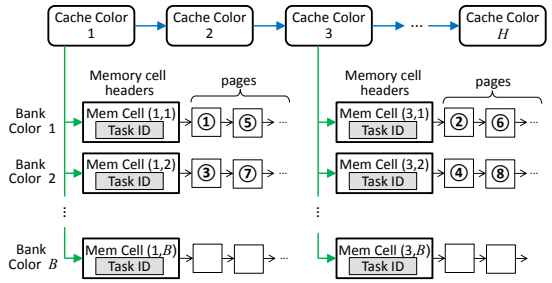

Fig. 5: Page management mechanism for combined cache and bank coloring

With these predicates, we define valid (meaning "valid allocation") as follows

Definition 3: valid($\mathcal{CA}$) =$(\forall \pi_j \in \pi : (\sum_{i \in [1,n] | P_i = j} \frac{C_i^{H_i}}{T_i} \leq 1)) \wedge (\neg conflict(\mathcal{CA}))$

Definition 4: valid($\mathcal{AA}$) =$(\forall \pi_j \in \pi : (\sum_{i \in [1,n] | P_i = j} \frac{C_i^{H_i}}{T_i} \leq 1)) \wedge (\neg conflict(\mathcal{AA}))$

The two previous definitions express that a valid allocation does not over-utilize any core[3] and has no conflict.

It is easy to see that any given abstract allocation can be transformed into a concrete allocation. Hence, our allocation algorithms (presented in Section IV) generate abstract allocations only.

E. OS Support for Combined Cache and Bank Coloring

In order to maintain physical pages according to their cache and bank colors, we developed a virtual page management mechanism with a two-level hierarchical list structure, as shown in Figure 5. Pages are first categorized according to their cache colors, and then they are sub-categorized according to their bank colors. The pages with the same cache and bank colors are linked as a list which represents a memory cell. The header of this list contains its cache color x and bank color y as (x, y). The header also maintains its owner task ID. Once a task is assigned its cache and bank colors, the task is restricted to use only physical pages in its own memory cells, thereby preventing cache and bank interferences from other tasks.

When a task is assigned more than one cache or bank color, we use a round-robin scheme to allocate pages to the task from its memory cells. This approach evenly distributes the task's page usage across multiple memory cells. In Figure 5, the circled numbers represent the page allocation order for a task that is assigned cache colors 1 and 3, and bank colors 1 and 2. Pages are allocated to this task from the memory cells of $(1, 1)$, $(3, 1)$, $(1, 2)$, and $(3, 2)$ in round-robin order.

We have implemented our page management mechanism as part of the memory reservation scheme [8] in Linux/RK [12], which is based on the Linux 2.6.38.8 kernel. Memory reservation allows an application task to reserve a portion of the physical memory for its exclusive use. Memory reservation maintains a global physical page pool and our page management mechanism is applied to this page pool. When a taskset is given, each task is assigned its cache and bank colors. Then, physical pages are reserved for each task from the page pool based on their assigned cache and bank color indices.

[3]In this paper we assume EDF scheduling.

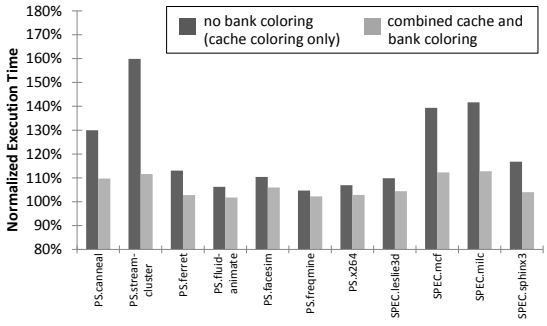

Fig. 6: Bank Interference Protection

F. Effect of Combined Cache and Bank Coloring

In this section, we evaluate how much temporal protection on memory accesses can be achieved by our combined cache and bank coloring approach. As a baseline, we use a subset of the PARSEC [3] benchmark (running one benchmark in each task) measuring their execution time when each task runs alone in the system. Then, we measured their execution times when a memory-intensive background task is co-executed on a different core. We compare the execution time increase due to the co-running task under our approach and the no-bank-protection (cache coloring only) approach.

The target system is equipped with the Intel Core i7-2600 3.4GHz quad-core processor. The system is configured for 4KB page frames and a 4GB memory reservation page pool. With cache and bank coloring, the system provides 32 cache colors and 16 bank colors. During this experiment, each task is assigned four exclusive cache partitions so tasks do not experience cache interference. Under our approach, each task is also assigned eight exclusive bank partitions. Under the no-bank-protection approach, the benchmark task and the background task share memory banks.

Figure 6 shows the execution time of each task when it runs with the background task. The execution times are normalized to the execution time when it runs alone in the system. Overall, the task execution time increase due to the background task is significantly lower under our approach. For instance, the execution time of `PS.streamcluster` is increased by 60% under the no-bank-protection approach, but the increase is only 12% under our combined cache and bank coloring approach. We suspect that this 12% of increase is caused by contention in other components of the DRAM system, such as the DRAM controller and the DRAM bus. The contention on such components can be mitigated by a memory bandwidth reservation mechanism [17] [2], but is beyond the scope of this work. We plan to study this issue as part of our future work.

IV. Coordinated Cache and Bank Color Allocation Algorithms

In this section, we present two algorithms for allocating tasks to processor cores and to non-conflicting cache and bank partitions. The first one is based on solving a Mixed-Integer Linear Program (MILP). It has the advantage that it is optimal but it has the drawback that its running time can be large. The second one is based on solving a variant of the knapsack problem. It has the advantage that it runs faster but it has the drawback that there are problem instances such that MILP can solve them but the knapsack algorithm cannot solve them.

A. Mixed-Integer Linear Programming Algorithm

From the previous discussion, it can be seen that our problem is to assign tasks to processors, assign cache colors to tasks and assign bank colors to processors so that certain constraints are fulfilled. It is therefore natural to phrase this as a constraint satisfaction problem. We will now present notations that we find useful and then present such constraints.

Let $\mathrm{PCS_i}$ denote the set of possible cache colors that we have specified that it is possible to assign to task τ_i. Consider for example a task τ_1 for which assigning 1 cache color to this task leads to the execution time 3.5 and assigning 2 cache colors to this task leads to the execution time 3.4 and assigning 6 cache colors to this task leads to the execution time 3.1 and assigning 7 cache colors to this task leads to the execution time 3.0. This gives us $\mathrm{PCS_i} = \{1, 2, 6, 7\}$ and $C_1^1 = 3.5$, $C_1^2 = 3.4$, $C_1^6 = 3.1$, $C_1^7 = 3.0$.

We compute $\mathrm{PC_i}$ as: $\mathrm{PC_i} = \{\ t : t \in \mathrm{PCS_i} \wedge (t \leq M_i) \wedge (t \leq H)\ \}$

Then finding an assignment of tasks to processors, assignment of cache colors to tasks and assignment of bank colors to processors can be done by finding an assignment of values to variables so that the constraints in Figure 7 are satisfied. Here, $x_{i,p} = 1$ indicates that task τ_i is assigned to processor p; otherwise $x_{i,p} = 0$. Also, $y_{i,t} = 1$ indicates that task τ_i is assigned t cache colors; otherwise $y_{i,t} = 0$. In addition, $z_{i,p,t} = 1$ indicates that task τ_i is assigned to processor p and task τ_i is assigned t cache colors; otherwise $z_{i,p,t} = 0$. Note that, for each task τ_i, we let $c_{i,p}$ indicate the amount of execution that task τ_i is assigned to processor p. Clearly, since a given task τ_i can only be assigned to a single processor, there is only one processor p such that $c_{i,p} > 0$; for the other processors, $c_{i,p} = 0$.

The constraints in Figure 7 use implication and equivalence operators. These can be rewritten to linear inequalities (using standard techniques) and hence the above is a Mixed-Integer Linear Program (MILP) — a problem for which many solvers are available (we use Gurobi [1]).

B. Knapsack Heuristic Algorithm

The main idea of our new knapsack heurstic algorithm is to (i) generate all possible assignments of banks to processor (line 3 in Algorithm 1), (ii) then generate task-to-processor assignment (see function `TaskKnapsack`) and (iii) finally, evaluate the feasibility of the current allocation against the capacity of the cache (line 6 in Algorithm 1). On each iteration k the call to `TaskKnapsack` returns an assignment candidate CTA_k. This candidate is composed of three sets: bank-to-core assignment set (BC_k), the task-to-core assignment set (TC_K), and the cache-to-task assignment set (HT_k).The BC_k set has one variable $B^k_{\pi_j}$ per core π_j with the number of banks assigned to the core. The set TC_k contains one variable P_i^k per task τ_i with the index of the core the task is assigned to. Finally, the set HT_k contains one variable H_i^k per task τ_i with the number of cache partitions assigned to task τ_i.

Algorithm 2 proceeds as follows. Cores are traversed in non-increasing order of number of assigned banks. In each iteration we try to maximize the total number of memory cells (required by the tasks) fitted in the memory grid. To do this we consider this total number of cells as the value of the objects

Constraints

$\forall p \in \{1,2,\ldots,m\} : \sum_{i=1..n}((1/T_i) * c_{i,p}) \leq 1$
$\forall i \in \{1,2,\ldots,n\} : \sum_{p=1..m} x_{i,p} = 1$
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\} : (x_{i,p} = 1) \Rightarrow (c_{i,p} = c_i)$
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\} : (x_{i,p} = 0) \Rightarrow (c_{i,p} = 0)$
$\forall i \in \{1,2,\ldots,n\} : \sum_{t \in \mathrm{PC_i}} y_{i,t} = 1$
$\forall i \in \{1,2,\ldots,n\} : c_i = \sum_{t \in \mathrm{PC_i}} (C_i^t * y_{i,t})$
$\sum_{p=1..m} \mathrm{ncachecolorsofproc_p} \leq \mathrm{H}$
$\sum_{p=1..m} \mathrm{nbankcolorsofproc_p} \leq \mathrm{B}$
$\forall p \in \{1,2,\ldots,m\} : \mathrm{ncachecolorsofproc_p} = \sum_{i=1..n} \sum_{t \in \mathrm{PC_i}} (t * z_{i,p,t})$
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\}, \forall t \in \mathrm{PC_i} : (z_{i,p,t} = 1) \Leftrightarrow (x_{i,p} = 1) \wedge (y_{i,t} = 1)$
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\}, \forall t \in \mathrm{PC_i} : (z_{i,p,t} = 1) \Rightarrow (\mathrm{M_i} \leq (\mathrm{nbankcolorsofprocessor_p} * t))$

Domains of variables

$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\} : x_{i,p} \in \{0,1\}$
$\forall i \in \{1,2,\ldots,n\} : \mathrm{c_i}$ is a real number ≥ 0
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\} : c_{i,p}$ is a real number ≥ 0
$\forall i \in \{1,2,\ldots,n\}, \forall t \in \mathrm{PC_i} : y_{i,t} \in \{0,1\}$
$\forall p \in \{1,2,\ldots,m\} : \mathrm{ncachecolorsofproc_p}$ is an integer in [1,H]
$\forall p \in \{1,2,\ldots,m\} : \mathrm{nbankcolorsofproc_p}$ is an integer in [1,B]
$\forall i \in \{1,2,\ldots,n\}, \forall p \in \{1,2,\ldots,m\}, \forall t \in \mathrm{PC_i} : z_{i,p,t} \in \{0,1\}$

Fig. 7: A MILP formulation for coordinated task, cache and bank allocation.

Algorithm 1 KnapsackCoreAndMemoryAssignment($\mathcal{S}$)

1: $CTA_k := (BC_k = \{B^k_{\pi_j} | \pi_j \in \pi\}, TC_k = \{P^k_i | \tau_i \in \tau \wedge \pi_{Pi} \in \pi\}, HT_k = \{H^k_i | \tau_i \in \tau\})$
2: /* CTA is an assignment candidate */
3: $BA \leftarrow$ a set of all possible candidate banks-to-core assignment
4: **for all** each assignment BA_k in BA **do**
5: $\quad CTA_k \Leftarrow TaskKnapsack(\tau, BA_k)$
6: $\quad$ **if** $\sum H^k_i \leq H$ **then return** CTA_k **end if**
7: **end for**
8: **return** null

(tasks) packed in the knapsack. We consider the total number cache partitions as the size of the knapsack and the cost of the object as the number of cache partitions required by each task τ_i (H_i). We calculate the cache partitions required by the task taking into account the bank partitions available in the core, i.e., $H_i = \left\lceil \frac{M_i}{B_{\pi_j}} \right\rceil$. Then, given this requirement and the total utilization of the tasks assigned to a core the algorithm searches for the best task-to-core allocations.

The tasks-to-core assignment exploration saves candidate assignments (to the current core) in a vector V with $H+1$ entries. Each entry v_k keeps the total utilization U_k of the current core, the total memory cells fitted RM_k (sum of memory assignments of all assigned tasks to the core), the set of task-to-core assignments TC_k, and the set of cache-to-task assignments HT_k. The array V represents possible packings for a knapsack of size up to H (indexed from 1 to H), with a special entry 0. The entry indexed by k represents a candidate that requires a total of k cache partitions. For example, when a candidate assignment Z consists of τ_x and τ_y, and both H_x and H_y are 2, the candidate assignment Z is held in an entry with index $2+2=4$ (v_4). An entry indexed by 0 represents an entry with no cache assignments. The main loop (Line 3) traverses all the cores. The next nested loop (Line 8) traverses all the tasks that have not been assigned to a core yet. Finally, the last nested loop (Line 10) traverses all candidate packing entries that requires k cache partitions. At iteration k, the algorithm tries to make a new candidate packing with the previous candidate packing in entry v_k by adding τ_i to the packing and checking its feasibility. The new cache requirement is calculated by adding to the old requirement k the cache requirement from τ_i (H^c_i). This new packing is stored with the index of the new assignment $k + H^c_i$, if we can verify that such requirement is less than or equal to H

Algorithm 2 TaskKnapsack(τ, BA)

1: $CTA_r := (BC_r = \{B^r_{\pi_j} | \pi_j \in \pi\}, TC_r = \{P^r_i | \tau_i \in \tau \wedge \pi_{Pi} \in \pi\}, HT_r = \{H^r_i | \tau_i \in \tau\})$
2: $BC_r \leftarrow BA$
3: **for all** $\pi_j \in \pi$ in non-increasing order of B_{π_j} **do**
4: $\quad$ // Vector of utilizations (U_k) mem reqs. (RM_k), task-to-core (TC_k) and
5: $\quad$ // cache-to-task (HT_k) index by number of cache assigned k
6: $\quad V := \{v_h = (U_h, RM_h, TC_h, HT_h) | 0 \leq h \leq H\}$
7: $\quad \forall i \in [0,\ldots,H]$: $U_i \leftarrow 0, RM_i \leftarrow 0$
8: $\quad$ **for all** $\tau_i \in \tau | P_i = 0$ **do**
9: $\quad\quad H^c_i \leftarrow \left\lceil \frac{M_i}{B_{\pi_j}} \right\rceil$
10: $\quad\quad$ **for all** $k \in [0,\ldots,H]$ **do**
11: $\quad\quad\quad$ **if** $((k = 0) \vee (k \neq 0 \wedge U_k \neq 0)) \wedge k + H^c_i \leq H \wedge \sum_{\tau_r \in \tau | P_r = j} \frac{C^{H_r}_r}{T_r} + \frac{C^{H^c_i}_i}{T_i} \leq 1$ **then**
12: $\quad\quad\quad\quad$ /* This candidate is feasible */
13: $\quad\quad\quad\quad$ **if** $RM_{k+H^c_i} < RM_k + M_i$ **then**
14: $\quad\quad\quad\quad\quad$ // New candidate can fit more memory cells
15: $\quad\quad\quad\quad\quad U_{k+H^c_i} \leftarrow U_k + \frac{C^{H^c_i}_i}{T_i}$
16: $\quad\quad\quad\quad\quad RM_{k+H^c_i} \leftarrow RM_k + M_i$
17: $\quad\quad\quad\quad\quad \forall P^k_x \in TC_k$: $P^{k+H^c_i}_x \leftarrow P^k_x$
18: $\quad\quad\quad\quad\quad P^{k+H^c_i}_i \leftarrow j$
19: $\quad\quad\quad\quad\quad \forall H^k_x \in HT_k$: $H^{k+H^c_i}_x \leftarrow H^k_x$
20: $\quad\quad\quad\quad\quad H^{k+H^c_i}_i \leftarrow H^c_i$
21: $\quad\quad\quad\quad$ **end if**
22: $\quad\quad\quad$ **end if**
23: $\quad\quad$ **end for**
24: $\quad$ **end for**
25: $\quad v_m = \mathrm{argmax}_{v \in V}(RM_i)$
26: $\quad TC_r \leftarrow TC_r \cup \{P^m_i \in TC_m | P^m_i = j\}$
27: $\quad HT_r \leftarrow HT_r \cup \{H^m_i \in HT_m\}$
28: **end for**
29: **return** $CTA_r = (BC_r, TC_r, HT_r)$

(i.e., it fits the knapsack). Similarly, the schedulability of the taskset is verified by checking that $U + \frac{C^{H^c_i}_i}{T_i} \leq 1$. Finally, if the new total number of cells is larger than the previous $RM_{k+H^c_i}$ this new packing is stored in the entry $v_{k+H^c_i}$, overwriting the previous contents, otherwise the candidate is discarded due to its lower value (lower fitting memory cells).

Once all tasks are assigned, we select the entry v_m in the vector V that maximizes the total number of cells RM_m deployed in the memory grid. From this entry we take the tasks assigned to core π_j and save these core assignments along with the assignments of cache partitions to the selected tasks. Then, the core loop continues until all cores are traversed.

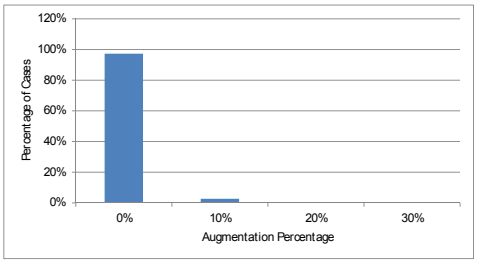

Fig. 8: Knapsack Augmentation Experiment Results

C. Allocation Algorithms Evaluation

We have generated problem instances randomly (see the appendix) and evaluated both algorithms.

1) Scalability: We conducted experiments with the following default setup: B=64, H=32 and m=4. Then we varied each of these parameters as follows. First, we varied m throughout all numbers in {4,8,16,32}. Then, we varied H throughout all numbers in {16,32,64,128}. Finally, we varied B throughout all numbers in {8,16,32,64}. For very large systems (the case that B=64, $H \geq 32$, m=4) MILP tended to not be able to finish within an hour. For other systems we evaluated, MILP tend to succeed.

2) Knapsack Augmentation Evaluation: Given that the knapsack algorithm can fail to allocate difficult cases we created an experiment that increases the size of the memory configuration and number of cores incrementally (augments the resource) until the algorithm succeeds. We ran 100 experiments with 4 cores, 16 cache colors, 32 bank colors, and 16 tasks. The results are presented in Figure 8. In this figure we can observe that in over 95% of the cases we can fit the taskset without augmentation and the rest of the cases can be fitted with only 10% augmentation.

V. Conclusions

We have presented a coordinated cache and bank coloring scheme that is designed to prevent cache and bank interference simultaneously. We also gave color allocation algorithms for configuring a virtual memory system to support our scheme which has been implemented in the Linux kernel.

References

[1] Gurobi. http://www.gurobi.com.

[2] F. Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical report TR-I4-97-02, University of Erlangen, Germany, 1997.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT'08*.

[4] D. Dasari, B. Andersson, V. Nélis, S. M. Petters, A. Easwaran, and J. Lee. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *ICESS'11*.

[5] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnosand, A. B. Nejad, A. Nelson, and S. Sinha. Virtual Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow. In *CRTS'12*.

[6] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010.

[7] H. Kim, A. Kandhalu, and R. Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real-Time Systems. In *ECRTS'13*.

[8] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *RTCSA'12*.

[9] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. PACT'12.

[10] M. Lv, G. Nan, W. Yi, and G. Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *RTSS'10*.

[11] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *RTAS'13*.

[12] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *RTSS-WIP'98*.

[13] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE'10*.

[14] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *CODES+ISSS'11*.

[15] J. Rosén, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *RTSS'07*.

[16] L. Steffens, M. Agarwal, and P. van der Wolf. Real-Time Analysis for Memory Access in Media Processing SoCs - A Practical Approach. In *ECRTS'08*.

[17] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality. In *ECRTS'12*.

Appendix

We generate the corresponding random taskset as follows.

1) First, generate a memory grid of HB memory cells.
2) Then, generate two sets of memory-division lines, one set to divide the banks and the other set to divide the caches. Each set has $M-1$ lines. These lines are generated randomly from a uniform distribution. These lines create M memory-grid "rectangles" g_k in the diagonal. These rectangles are the non-intersecting regions to be used by each of the k cores. Each rectangle has a number of cache colors $ccolors(g_k)$ that represents its size in the cache dimension of the grid and a number of banks colors $bcolors(g_k)$ that represents its size in the bank dimension of the grid.
3) For each core k, generate a taskset of size N_k ensuring that $N_k \leq ccolors(g_k)$.
4) For each core k, generate $N_k - 1$ cache-division lines to divide the core rectangle g_k.
5) Assign a random number of memory cells $h_{i,k}$ to a each task $\tau_{i,k}$ in a core k such that $\sum_{\forall i} \left\lceil \frac{h_{i,k}}{bcolor(g_k)} \right\rceil \leq ccolor(g_k)$.
6) For each task τ_i, randomly generate the T_i from a uniform distribution between $[100, 2000]$.
7) For each task τ_i, generate C_i^t as

$$C_i^t = \frac{0.99}{N_k} T_i \cdot \ (1 - r_i) + \frac{0.99}{N_k} T_i \cdot r_i \frac{1}{t}$$

where r_i is a randomly generate number in the range $[0, 0.5]$.