Decoupling Translation Lookaside Buffer Coherence from Cache Coherence

Hao Liu, Quentin L. Meunier, Alain Greiner

Decoupling Translation Lookaside Buffer Coherence from Cache Coherence

Hao Liu, Quentin L. Meunier and Alain Greiner
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, 4 place Jussieu 75005 Paris.
Emails: {hao.liu,quentin.meunier,alain.greiner}*@lip6.fr*

***Abstract*—Many multicore and manycore architectures support hardware cache coherence. However, most of them rely on software techniques to maintain Translation Lookaside Buffer (TLB) coherence, namely the TLB shootdown routine, which is a costly procedure, known to be hardly scalable.**

The TSAR architecture is a manycore architecture including hardware TLB coherence, but in which the TLB coherence mechanism is tightly coupled to the cache coherence protocol, resulting in useless TLB invalidations. We propose to improve this existing TLB coherence scheme by adding a hardware module which allows separating data from metadata for cache lines containing address translation. This allows to eliminate the need to invalidate TLB entries when a line containing a translation is evicted from the L1 cache.

Our solution does not modify the cache coherence protocol, does not increase the critical path in the L1 cache, and even results in little memory savings. Performance results show that our solution allows to eliminate from 90% to 95% of TLB scans operations, and from 50% to 80% of TLB flushes. This in turn results in an overall performance improvement of 5% to 20% of execution times on a 16-core architecture.

***Index Terms*—Translation Lookaside Buffer; Hardware TLB Coherence; TLB Shootdown; TLB Flush; Manycore; Scalability;**

1. Introduction

Thanks to technology advances, we can put an ever growing number of cores in one single chip [1], leading to so called multi- and manycore chips [2]. If the question of coherence is not definitely answered for these architectures, it is now rather widely accepted that hardware supported on-chip coherence will be part of future architectures [3], to the detriment of software coherence.

However, most manycore architectures [4], [5], [6] still rely on the operating system to maintain TLB coherence. TLB coherence refers to the fact that when there is a modification of a piece of data which is a page table entry, namely an address translation, it is necessary to invalidate or update the copies of this translation in all the TLBs (both instruction and data). If not done properly, the other threads of the process with outdated page table entries in their TLB can access invalid pages.

The most common approach for dealing with TLB coherence is to use the TLB shootdown approach [7]. In this approach, the operating system is in charge of sending an invalidation interrupt to all the processors owning the translation. However, this approach is not convenient when TLB misses are handled in hardware: since the operating system has no knowledge of the location of the copies to invalidate, it must use a broadcast policy and interrupt all the threads. As shown in [8], this approach is both inefficient and hardly scalable, suggesting that hardware TLB coherence will emerge as a standard solution in future chips, following the same evolution as cache coherence.

In this article, we do not try to demonstrate that hardware TLB coherence is better than software coherence, but instead focus on the link between cache coherence and TLB coherence: we show that a decoupled coherence scheme is beneficial for performances compared to a unified cache and TLB coherence scheme. Section 2 describes the problematic of maintaining TLB coherence in hardware and presents related works; section 3 details our proposed solution; section 4 describes our experimental results, while section 5 concludes.

2. Hardware TLB Coherence

There have been a various number of techniques proposed for maintaining TLB coherence in hardware in a multiprocessor system. [9] presents a technique for maintaining TLB coherence in an I/O bridge. The coherence protocol guarantees an exclusive access to the page table containing the entry placed into the TLB, thus forbidding the sharing of the page with other I/O bridges (or processors). As such, this solution is not applicable when there are many threads to run in parallel, all sharing the same page table, what can be expected on a manycore architecture.

In [10], a mechanism is introduced for maintaining TLB coherence based on specific messages which are broadcast on every TLB miss, TLB eviction or translation modification. Not many details are given about the protocol or the architecture itself, but the systematic broadcast strategy is not sustainable for manycore architectures; besides, a particular protocol has to be defined for specific TLB communication.

[11] uses a *poison* bit associated with each memory block, which is set when the virtual to physical translation for this block is modified. An exception is then generated as a response to a query related to this translation. We find that the main drawback of this approach is the way the poison bit has to be reset, which requires flushing the TLBs and timestamping these flushes, what is very costly.

978-1-5090-6762-6/17/$31.00

[12] uses a L2 directory for TLB, keeping a bitmap for all entries. When the OS sends a shootdown, this directory is accessed to only sends invalidations to the corresponding TBLs. This approach does not fit our hypotheses: (a) it uses a bitmap which does not scale for a manycore architecture; and (b) it does not eliminate the intervention of the operating system.

The TSAR architecture [13], [14], used as a baseline for this work, follows the same principles as the approach presented in [8]. In this approach, TLB coherence is *plugged* to the cache coherence protocol, and both the protocol between L1 and L2 caches and the L2 caches themselves remain unmodified.

The rest of the section presents the baseline scheme for maintaining TLB coherence. We assume an architecture with 32-bit virtual addresses and 40-bit physical addresses, maintaining coherence between L1 and L2 caches in hardware. Each L1 cache is split between instruction and data, and there is one TLB for each part, composed of 8 ways and 8 sets. The coherence protocol uses a write-through strategy, but the solution for TLB coherence could be applied for any type of cache coherence protocol.

In case of a TLB miss, a hardware automaton (called *Table-Walk*) is in charge of accessing the page table located in memory and updating the TLB accordingly. The TLB coherence in our baseline system is maintained in hardware, using the following rules:

- Page tables are cacheable;
- In case of TLB miss, the Table-Walk automaton must compute the Page Table Descriptor (PTD) address, and then the Page Table Entry (PTE) address;
- The Table-Walk automaton does not directly send requests to the memory but transmits them to the local data cache;
- If the local L1 cache contains the page table line containing the missing translation, it gives it to the TLB which is updated. Otherwise, the data cache controller gets it from the L2 cache;
- Since a coherence protocol is maintaining coherence between L1 and L2 caches, any modification of the page table will be propagated in the L1 caches owning a copy;
- L1 caches maintain for each directory entry a `is_ppn` bit indicating whether the line belongs to a second level page table, and thus may contain Physical Page Numbers (PPNs) which have been copied into TLBs;
- Similarly, L1 caches contain for each directory entry a `is_ptn` bit indicating that the line belongs to a first level page table, and thus may contain PTDs which have been used to compute PPNs stored into TLBs;
- Every TLB entry contains a `nline` field, which is the cache line number of the line containing the virtual to physical translation.
- If the L1 cache evicts a line `line` having the `is_ppn` bit set, it performs an associative search in both the ITLB and DTLB to invalidate the entries contained in the line `line` (operation *Scan-TLB*);
- If the L1 cache controller evicts a line having the `is_ptn` bit set, it performs an invalidation of all entries in both TLBs (operation *Flush-TLB*), since any translation could be in the second level page table of the modified first table entry.

For the last two items, the line eviction can be caused by two distinct reasons: either the L1 cache needs to free some space in order to store the result of a miss (local eviction), or it has received a coherence request on the considered line.

Although a prototype ASIC version of the TSAR architecture is currently built with this baseline scheme, the latter has a drawback since it imposes the strong inclusivity of both TLBs in the L1 cache, in order to guaranty that future coherence requests will be forwarded. Indeed, updates or invalidations targeting a line containing one or several translations can be made only if the L2 cache is aware that these translations exist. This implies that any eviction from the L1 cache of a line marked either `is_ppn` or `is_ptn` (later referenced as `in_tlb`) requires to invalidate all corresponding TLBs entries: if not done, it is possible that a page table update will not be seen by this cache, resulting in an invalid access, for example after a page unmap.

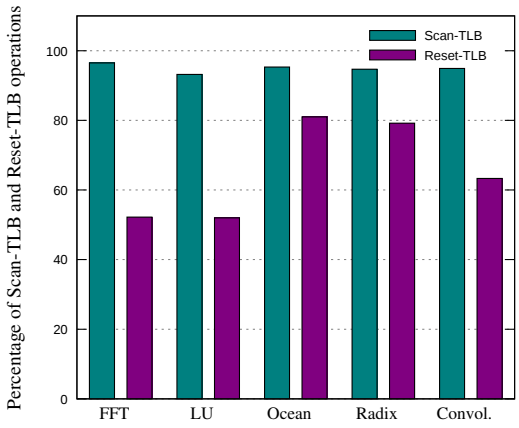


Figure 1: Percentage of *Scan-TLB* and *Flush-TLB* operations caused by local evictions

The TLB operations resulting from local evictions harm performance, since local evictions are responsible for approximately 50% of *Flush-TLB* and 95% of *Scan-TLB* operations, as shown in figure 1. The contribution of this article is to propose a new hardware TLB coherence scheme, which releases this inclusivity constraint, thus allowing a TLB translation to remain valid when the corresponding cache line is locally evicted.

One idea could be to promote the lines marked `in_tlb` when a local eviction happens. However, it is not possible to update the LRU information depending on the translations accessed, since it would require to access the cache twice for each instruction or data access: a first time for the effective instruction or data, and then after the translation and retrieval of the line containing this translation, a second time to update the LRU information of that line. Thus, such a policy implies having a fixed higher priority for lines marked `in_tlb`, with the risk of polluting the cache endlessly with useless translations.

Instead, our solution consists in decoupling metadata associated to lines containing translations from their presence in the L1 cache.

3. Proposed Solution

3.1. Principles

When a page table entry is modified by the OS, the write-through strategy of the coherence protocol is used: the line is first updated in the local L1 cache if it is present; then the write is propagated to the L2 cache which sends invalidations or updates to caches containing copies. When such a request arrives at a L1 cache, the corresponding TLB entries are invalidated as before via a *Scan-TLB* or *Flush-TLB* operation.

In order to maintain coherence, a L1 cache needs to continue to receive coherence requests for evicted lines corresponding to translations which are still present in a TLB. This means that for evicting such a line, the eviction must be silent and not notified to the L2 cache. This is done by storing information associated to these lines outside from the cache, in order to treat properly coherence requests.

For this, our proposed solution introduces a hardware structure, called PT^3, for *Persistent TLB Translation Table*, which allows to decouple metadata associated to lines containing translations from their presence in the L1 cache. Upon reception of coherence requests, the PT^3 is accessed along with the cache to check if the target line contains translations.

Our solution integrates into the existing coherence protocol, as is the case in [8]; however, the latter solution must access a content addressable memory for every write, which costs several cycles, and could potentially block the processor in case of several consecutive writes. On the contrary, our solution first tests if a write targets a TLB entry with no additional cost, and then performs the associated operations in the subsequent cycles, but only when the test returned true.

3.2. Structure of the PT^3

The PT^3 is essentially a metadata set-associative cache for lines marked `in_tlb`, storing for each of its line, the number of corresponding entries contained in the ITLB and DTLB. It contains 8 sets of 8 ways each.

The structure of a PT^3 entry is described in table 1. When a cache line X marked `in_tlb` is locally evicted from the L1 cache, the entry bit `in_cache` is reset, and the associated TLB entries remain valid. The L1 cache controller does not send a *cleanup* request to the L2 cache. In case of a TLB miss, the L1 cache is searched to obtain the PPN and the PT^3 is accessed to know whether it already contains an entry for this line X. Four cases are possible:

- If the line is both valid in the cache and in the PT^3, the `count` field in the PT^3 is incremented (`count` ← `count` + 1).
- If the line is valid in the cache but not in the PT^3, a new entry for this line is added in the PT^3 (`count` ← 1, `in_cache` ← 1).

TABLE 1: Structure of a PT^3 entry

Field name	Size	Description
`nline`	31 bits	TAG of the physical address of the line in the PT^3
`valid`	1 bit	Valid bit
`lru`	1 bit	LRU bit for victim selection
`count`	5 bits	Number of TLB entries matching the line (max. 16 entries per line)
`in_cache`	1 bit	Set if the matching line is present in the data cache
`kernel`	1 bit	Set if the line contains entries of the kernel page table
`ptd`	1 bit	Set if the matching line contains at least one entry with the `is_ptn` bit set

- If the line is absent from both the L1 cache and the PT^3, a *read* request is sent to the L2 cache, and upon response a new entry is created in the PT^3 (`count` ← 1, `in_cache` ← 1).
- If the line is invalid in the cache but valid in the PT^3, this means that the line has been evicted from the L1 cache but that the L2 cache has not been informed of this eviction. An *uncached read* request is sent to the L2 cache, which does not modify the line state in the L2 cache directory. The L1 cache and the PT^3 entry are updated (`count` ← `count` + 1, `in_cache` ← 1).

When the L1 cache receives an *update*, receives an *invalidate*, or emits a *write* request for a line marked `in_tlb`, the operation *Scan-TLB* or *Flush-TLB* is performed only if the `count` field value is not 0.

In case of a context switch, all the TLBs entries need to be invalidated, except the entries corresponding to operating system pages, since these pages are replicated in all processes; these entries are identified with a `kernel` bit in the TLBs. Similarly, the entries in the PT^3 with the `kernel` bit set are kept valid, while the others are reinitialized (`count` = 0) but not invalidated. Indeed, if the bit `in_cache` is 0, the invalidation of the entry requires the sending of a *cleanup* request. To avoid the sending of potentially many *cleanup* requests, the entries with `count` = 0 will instead be lazily replaced when necessary.

The whole cache structure, including the TLB and the PT^3, is shown in figure 2.

3.3. Hardware Cost Comparison

The number of bits memorized in the PT^3 is 64 lines $\times$ 41 bits per line, i.e. a total of 2,624 bits.

However, since the PT^3 contains all the information relative to TLB coherence, some fields in both L1 caches and TLB can

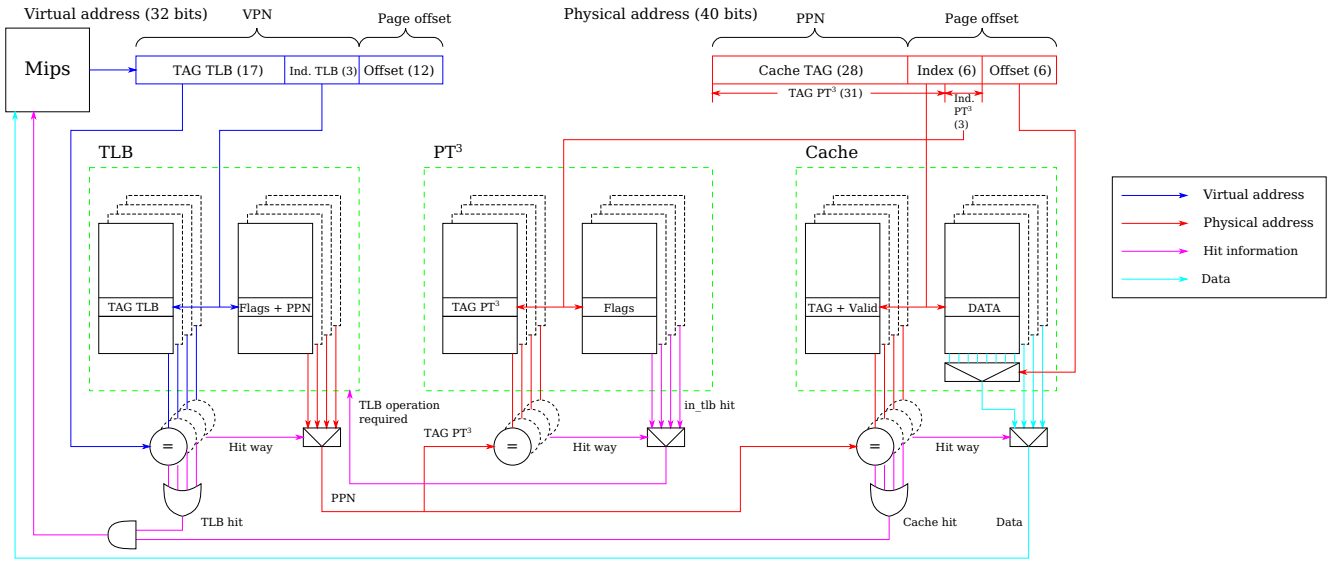


Figure 2: L1 Cache Structure with the PT^3

be removed. In particular, the two bits `is_ppn` and `is_ptn` from each directory entry are now useless, resulting in 256 × 2 = 512 bits saved; in the TLB, the 34-bit wide `nline` field can be replaced with a 6-bit PT^3 offset, resulting in 2 × 64 × (34 - 6) = 3,584 bits saved compared to the baseline solution. Overall, the number of bits saved thanks to this solution is 1,472.

Of course, additional logic is necessary in the L1 cache, but this overhead is expected to be almost negligible compared to the whole L1 cache controller.

3.4. Victim Selection

The small size of the PT^3 implies that a valid entry can be evicted to add a new entry instead. The victim selection process aims at minimizing the cost of an eviction; more precisely, lines are searched in the following order, by increasing cost:

- An invalid entry;
- An unused entry (`count` = 0) such that `in_cache` = 1. The cost of this eviction is null;
- An unused entry (`count` = 0) such that `in_cache` = 0. A *cleanup* request must be sent to the L2 cache;
- An entry containing no PTD (`ptd` = 0). A *Scan-TLB* operation is required to invalidate all TLB entries matching the selected line;
- An entry containing a PTD (`ptd` = 1). A *Flush-TLB* operation is required on both TLBs.

3.5. Critical Path Delay Analysis

The PT^3 does not increase the L1 cache critical path. In the L1 controller, the critical path consists in accessing the data part of the data cache (in SRAM) and responding to a processor request before the end of the cycle, as illustrated in figure 3. The TLB (in registers) can be accessed at the same

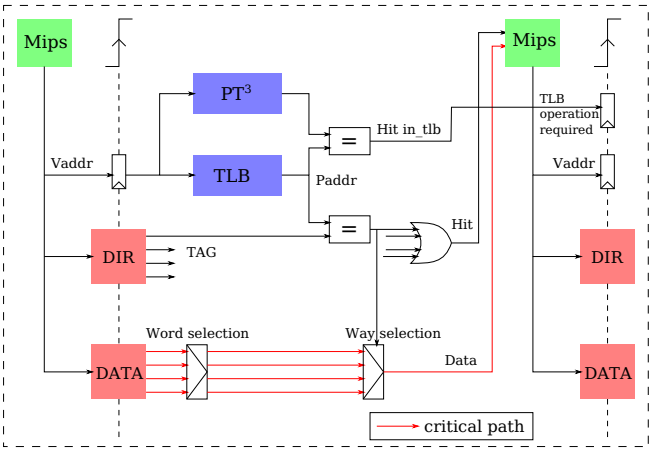


Figure 3: Critical Path in the L1 Cache

time as the L1 cache since the number of bits to address the cache (12) is equal to the number of bits for the page offset. Accessing the PT^3 can also be made in parallel with the other accesses since the bits used to index it are contained in the page offset, and its structure is very similar to the TLB. This access is required for every write to detect if a subsequent TLB operation is necessary.

3.6. Working Example

Table 2 presents a scenario example showing the evolution of information inside the TLB, PT^3 and L1 cache to better understand the interactions between these components.

The events occurring are:

TABLE 2: Working Example for the PT^3

	TLB	**PT^3**			**L1 Cache**
Event	Number of entries	`valid`	`count`	`in_cache`	`valid`
a	0	0	0	0	0
b	0	0	0	0	1
c	1	1	1	1	1
d	2	1	2	1	1
e	2	1	2	0	0
f	3	1	3	1	1
g	0	0	0	0	0
h	1	1	1	1	1
i	0	1	0	1	1
j	0	0	0	0	0

(a) At startup, no component contains any information on line L.
(b) Line L is accessed by the operating system to build the page table. It is copied in the L1 cache.
(c) A TLB miss happens, for which the Table-Walk automaton finds the PPN in line L. The PT^3 stores information about line L in a new entry (`count` = 1, `in_cache` = 1). The missing PPN is stored in the TLB along with the corresponding index in the PT^3.
(d) Another TLB miss happens which has its translation in line L. The entry for line L in the PT^3 is updated: `count` = 2. The missing PPN is stored in the TLB, along with the index in the PT^3.
(e) Line L is evicted from the L1 to free some space. The field `in_cache` is decreased to 0 in the PT^3. The L1 cache does not send a *cleanup* request to the L2.
(f) A third TLB miss happens. Line L is fetched again in the L1 cache. The missing PPN is stored in the TLB with the PT^3 index. The TLB now has 3 valid entries for line L. In the PT^3, `count` = 3 and `in_cache` = 1.
(g) The L1 cache controller receives an *invalidate* request for line L. It performs a *Scan-TLB* operation in order to invalidate all 3 entries corresponding to L in both TLBs. It invalidates line L in the L1 cache and sends a *cleanup* request to the L2.
(h) After a new TLB miss, the 3 components are in the same state as step (c).
(i) The TLB evicts the entry included in line L to replace it. In the PT^3, the `count` field for this line is set to 0, but this line is still present in the L1 cache.
(j) The L1 cache controller receives an *invalidate* request. This time, the controller invalidates line L in the cache and sends a *cleanup* request to the L2, but does not send a *Scan-TLB* operation since the `count` field for this line is 0.

4. Experimental Results

We implemented our solution in the cycle-accurate SystemC model of the TSAR architecture [13], extending the SoCLib library [15]. We compare both solutions on the following metrics: total execution time, number of TLB misses and number of cache misses caused by a TLB miss. We also checked that the PT^3 allowed to make the number of TLB operations fitting results shown in figure 1: since all operations related to local evictions are removed, we checked that TLB operations related to writes or coherence request did not increase, thus resulting in a drop of 90% to 95% of *Scan-TLB* operations and of 50% to 80% of *Flush-TLB* operations.

The architecture and application parameters are described in table 3. Applications are taken from Splash-2 [16], except for Convolution, which is a convolution filter applied to an image. We selected these applications w.r.t. their reasonable simulation time and present all the results we obtained. Note that these applications do not particularly map or unmap a lot of pages, as the effect we want to measure, cache and TLB coherence coupling, can be visible even with no TLB modification. The OS we used is NetBSD, with a manual placement of threads on cores.

TABLE 3: Simulation Parameters

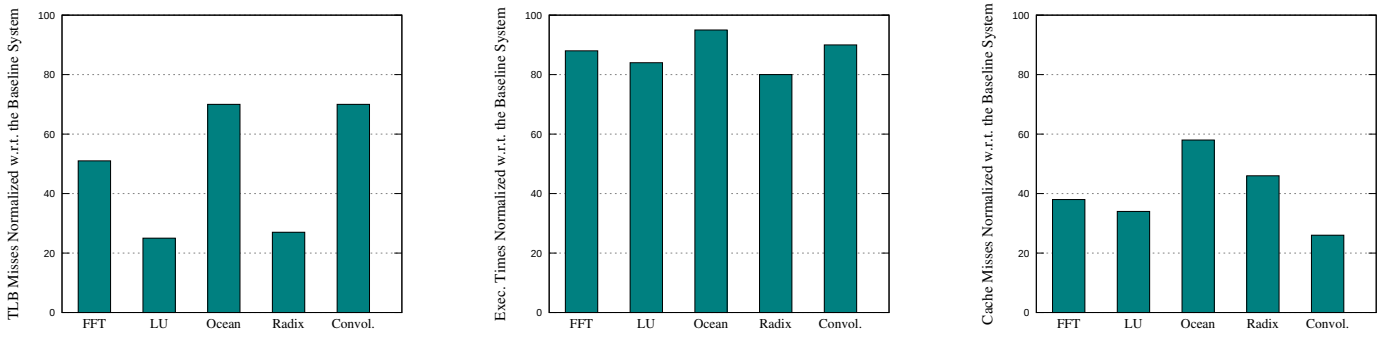
(a) Architecture Parameters

Processor type	Mips-32
Mesh Size (1 core/node)	4×4
L1 Cache Sets (I & D)	64
L1 Cache Ways (I & D)	4
L1 Cache Words (I & D)	16
L2 Cache Sets	256
L2 Cache Ways	16
L2 Cache Words	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8
PT^3 Sets (I & D)	8
PT^3 Ways (I & D)	8

(b) Applications Parameters

Benchmark	Input Data
FFT	2^{10} points
Radix	262,144 keys
LU	512×512 elements
Ocean	258×258 elements
Convol.	1024×1024 image

Results are presented in figure 4, and are normalized w.r.t.



(a) TLB Misses (b) Execution Times (c) Cache Misses caused by TLB Misses

Figure 4: Results with the PT^3 Normalized w.r.t. the Baseline Architecture

values on the baseline architecture.

Figure 4a shows that the number of TLB misses is largely reduced, reaching 70% for LU and Radix. Figure 4b shows that the total execution times are reduced from 5% to 20%, corroborating the TLB misses results. Finally, we analyze the number of cache misses caused by TLB misses. The PT^3 reduces these misses by approximately 60%. This result is twice positive: first, it decreases the network traffic; second, it minimizes the contention in the L2 cache containing the page tables. We also expect the decrease in traffic to turn into overall energy savings, although we did not perform such measurements.

5. Conclusion

We proposed an efficient solution for maintaining TLB coherence in hardware, and we advocate that TLB coherence should not be coupled with cache coherence, since it can lead to useless TLB invalidations. The main idea consists in releasing the inclusivity constraint between TLBs and L1 caches, by storing coherence related information in a separate structure called PT^3. Moreover, this structure indirectly increases the L1 cache capacity, since it allows not to duplicate in the L1 cache the information stored in the PT^3. Finally, this solution has a better hardware memory cost than the original one, and it does not increase the critical path in the L1 cache.

We applied this solution to the cycle-accurate model of the TSAR manycore architecture, and observed that this solutions improves the hit rate on both TLBs. This translates into performance improvements on the total execution time, ranging from 5% to 20% on five applications.

Future work includes the implementation of our solution at the RTL level in order to integrate it in the reference VHDL TSAR models. This will allow for measurements of the energy saved by this solution.

References

[1] J. M. Rabaey, "Scaling the power wall: Revisiting the low-power design rules," *Keynote speech at SoC*, vol. 7, 2007.

[2] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 746–749.

[3] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[4] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: a 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 477–488.

[5] C. Ramey, "Tile-gx100 manycore processor: Acceleration interfaces and architecture," *Tilera Corporation*, 2011.

[6] A. Ros, M. E. Acacio, and J. M. Garcıa, "Cache coherence protocols for many-core cmps," *Parallel and Distributed Computing*, 2010.

[7] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, *Translation lookaside buffer consistency: a software approach*. ACM, 1989, vol. 17, no. 2.

[8] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "Unified instruction/translation/data (unitd) coherence: One protocol to rule them all," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[9] S. Duncan, "Coherent translation look-aside buffer," 2003, uS Patent 6,633,967.

[10] P. Damron, "Method and system for translation lookaside buffer coherence in multiprocessor systems," 2005, uS Patent 6,931,510.

[11] J. Laudon and D. Lenoski, "System and method for maintaining coherency of virtual-to-physical memory translations in a multiprocessor computer," 2001, uS Patent 6,182,195.

[12] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 340–349.

[13] *TSAR: Tera-Scale Multiprocessor ARchitecture*, Available: https://www-soc.lip6.fr/trac/tsar, 2010.

[14] Y. Gao, "Generic cache controller for a massively parallel manycore architecture using coherent shared memory," Ph.D. dissertation, Université Pierre et Marie Curie (UPMC), 2011.

[15] The Soclib Consortium, "Soclib: an open platform for virtual prototyping of multi-processors system on chip," [Online]. Available: http://www.soclib.fr, Tech. Rep., 2008.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. New York: ACM Press, 1995, pp. 24–37.