Accurate Private/Shared Classification of Memory Accesses: A Run-time Analysis System for the LEON3 Multi-core Processor

Nam Ho, Ishraq Ibne Ashraf, Paul Kaufmann and Marco Platzner
Department of Computer Science, University of Paderborn, Germany
Email: namh@mail.upb.de, paul.kaufmann@gmail.com, platzner@upb.de

***Abstract*—Related work has presented simulation-based experiments to classify data accesses in a shared memory multi-core into private and shared. This information can be used to selectively turn on/off cache coherency mechanisms for data blocks, which can save memory bus bandwidth, minimize energy consumption, and reduce application runtimes. In this paper we present an implementation of a private/shared classification mechanism on a LEON3 SPARC multi-core processor running the Linux 2.6 kernel. Our mechanism is paged-based and allows for classifying and counting data accesses at run-time. Compared to previous work, our system provides more accurate, i.e., realistic, data as it includes a real multi-core architecture and an OS. Additionally, our prototype allows us to quantitatively evaluate the overhead for the classification mechanism. We test our system with sequential and parallel benchmarks from the Mibench, ParaMibench, PARSEC, and SPLASH2 application suites. The results show that parallel benchmarks are promising targets for selectively controlling coherency mechanisms and that the run-time overheads induced by our mechanism are rather small.**

I. Introduction

Modern shared memory multi-core processors usually implement one to two levels of private cache for each of the processing cores and one to two levels of shared cache. To achieve memory coherency, hardware-based cache coherence protocols are used. In a snooping-based protocol all outermost private caches are connected by a shared communication medium such as a bus or switch. Every outermost private cache holding a copy of a data block can track the sharing status of this block by monitoring the communication medium. This could involve, for example, broadcasting invalidation messages to the cache controllers of other core's private caches. Related work has argued that pessimistic strategies for activating the coherency mechanism, i.e., activating the mechanism although it is not required, may lead to significant computational and energy overheads [1]–[3]. The reported numbers for sequential and parallel applications revealed that on average 75% to 79% of the modified cache blocks are privately cached only by a single core [3], [4]. Turning off coherency for these cache blocks can save system bus bandwidth, minimize energy consumption, and reduce application runtimes.

In this paper we present a run-time classification system that detects whether memory accesses hit a private or a shared page. The novel contributions of this work are:

1) We present a `private/shared` memory page classification system build into the Memory Management Unit (MMU) of a LEON3 multi-core processor implemented on a Xilinx Virtex 6 FPGA. Our implementation comprises both LEON3 hardware extensions and extensions to the Linux 2.6 OS kernel, where we have added interrupt handlers for managing and controlling `private/shared` monitoring during run-time.
2) We provide a definition for `private/shared` memory access classifications and leverage the performance monitoring unit (PMU) module of [5] to integrate `private/shared` event counters into the standard Linux `perf` monitoring system for micro-architectural events.
3) We evaluate our system extensively regarding the computational overheads introduced by the `private/shared` classification system based on a representative set of sequential and parallel applications.

To the best of our knowledge, this work is the first to prototypically implement a `private/shared` memory access classification. Compared to previous simulation-based approaches our work provides more accurate, i.e., realistic, data as it includes a real multi-core architecture and an OS. Additionally, the prototype allows us to quantitatively evaluate the overhead for the detection mechanism, which is important to judge the practicability of a system that switches on and off the cache coherency mechanism selectively for data blocks at run-time.

The remainder of the paper is structured as follows: Section II provides an overview over related approaches. Section III discusses basics of `private/shared` data classification and includes our definition of these terms. The system architecture of our implementation including our extensions to the LEON3 multi-core and to the Linux 2.6 OS kernel are elaborated on in Section IV. Section V presents result from our experiments with sequential and parallel benchmarks and, finally, Section VI concludes the paper.

II. Related Work

Noticeable work on data classification was presented by Hardavellas et al. [3], who demonstrated it as one of the key requirements for the implementation of an efficient data placement policy for shared distributed Last Level Caches

(LLCs) in tiled architectures. In such an architecture the latency of an LLC hit depends on the physical location of the requested data in cache memory, which is partitioned into slices distributed among tiles. The authors propose putting data blocks classified as private into slices at the requesting cores to achieve minimal access latencies and avoid triggering the coherence mechanism. Shared data blocks accessed by multiple cores should be placed at a fixed slice to simplify the coherency mechanism. The central observation of the paper is that 72% to 90% of memory accesses are private and that switching off coherency may have a significant impact on the coherency overhead for these data. This potential has been investigated in follow-up work [1], [2] by proposing strategies for temporarily deactivating the coherence mechanism.

Other publications on `private/shared` classification of memory accesses considered energy minimization for caches [6], improved system performances [7], [8], reduced latency of virtual-to-physical-address translation [9], and simplification of multi-core coherence protocol design [10].

All related work approaches presented so far, except [7], act at page granularity, where whole pages and the page accesses are classified either as `private` or `shared`. The classification bits are usually stored in Page Table Entries (PTEs) and the OS is responsible for managing and updating their status. For such an OS-based approach the hardware overhead is negligible. Pure hardware-based approaches were proposed in [4], [11]–[13]. Although they can avoid OS modifications, the area requirements are significantly increased. For example, direct TLB-to-TLB communication channels are required for exchanging classification information among cores [4], [11] and dedicated hardware modules are needed for directory-based systems [12], [13].

An *a-priory* approach based on static code analysis was proposed by Li et al. [7]. The authors analyze memory accesses of multi-threaded applications and predict cores that will potentially share variables. Inherent to such an analysis technique is, however, that its accuracy depends heavily on the initial conditions, which have to be estimated, and on the dynamics of OS scheduling.

While our approach shares the page-level granularity of `private/shared` classification with most of the related work, it differs in the following aspects: First, with the integration of `private/shared` monitoring into a real processor implementation and OS our experiments deliver accurate classifications compared to previous simulation-based approaches that involve abstractions of the processor architecture and/or the OS. Second, we are able to quantify the computational overhead introduced by the `private/shared` detection and classification.

III. Private/Shared Classification

When classifying data as private or shared two questions arise: What is a meaningful data block granularity and when is a data block actually considered as shared data? While one can imagine to tag single bytes, words, and even arbitrary-sized memory blocks as `private` or `shared`, from the perspective of processor architecture a cache block is the smallest reasonable granularity. A cache block is the smallest chunk of data a processor usually transfers to and from the main memory in one step. Thus, even if a smaller granularity was chosen, for example by allowing a cache block to have multiple regions with different `private/shared` categorizations, we would still have to handle the whole cache block when resolving incoherencies. The next larger canonical granularity for `private/shared` classification is a memory page. Paging provides a partitioning of physical and virtual memory address spaces into blocks of usually the same size for an efficient management of virtual address spaces. Tagging pages `private/shared` is the most popular approach in related work as the implementation overheads for the MMU and OS are relatively small. The main consideration for the page-based approach is that a major part of a shared page may contain data accessed exclusively by a single core, thus being private and unnecessarily processed by the coherency mechanism. In fact, it is reported that on average only 6% to 26% of a shared page is accessed by more than one application, but that accesses to shared data dominate [3]. Generally, we are faced with a tradeoff between complexity and overhead of the classification mechanism on one hand, and the effort needed for maintaining coherence on the other hand. In our work we select page-level granularity for `private/shared` tagging to be able to efficiently implement the hardware and software memory monitoring mechanisms and compare the results to observations of related work.

Another question that determines how often the coherency mechanism needs to be triggered is when a page is actually shared. Let us assume a typical virtual memory system with a page-based `private/shared` classification implemented by storing `private/shared` bits in the PTEs and, consequently, also within the TLB entries. Every write access will consult the TLB to check the `private/shared` status of the addressed data, which requires the L1 cache to be virtually or physically tagged and physically indexed. From an OS point of view, if two processes `A` and `B` access the same page in read/write mode, the page is usually considered shared. However, if `A` and `B` execute on the same core, there is no need for checking coherency and the page could be considered `private` from the perspective of the coherency mechanism (cf. Fig. 1 (a)). Another issue is migration between cores. For example, if a private page `X` was modified by a process `A` on core 1, migrating `A` to core 2 renders `X` shared as `X` is now accessed by a different core with separate caches (cf. Fig. 1 (b)).

Generally, the definition of `private` and `shared` depends on the tradeoff between the complexity and overhead of classifying pages and the effort for maintaining coherence. For instance, the problem of pages becoming shared on migration of a process can be resolved by updating the PTEs of the process during the migration. As refining the `private/shared` classification can be done in many different ways, we emphasize simplicity in this work and select a canonical definition categorizing memory accesses as:

- *Private access:* refers to memory references to pages, which are being requested by only one processing core.
- *Shared access:* refers to memory references to pages, which have received requests from more than one processing core.

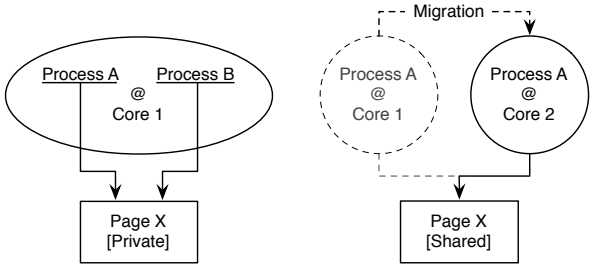


Figure 1: Private/shared classification of a page. Fig. (a): Page `X` can be tagged private since processes `A` and `B` execute on the same core. Fig. (b): On migration of process `A`, its previously private page `X` could become shared.

IV. System Architecture

Our design of a run-time `private/shared` classification mechanism consists of a hardware and a software part. The hardware implementation extends the PMU and MMU of the LEON3 softcore processor for counting accesses to private and shared pages as well as for detecting and signaling of initializations and re-classifications of page shared states to the CPU. The software implementation consists of two interrupt handlers in the Linux 2.6 kernel that initialize and re-classify the shared status of new and existent PTEs, respectively.

A. Extending the LEON3 Design

Our extensions of the LEON3 quad-core architecture are presented in Fig. 2. The default components of LEON3 are an Integer Unit (IU), a split Level-1 instruction and data cache (L1:I/D), and an MMU. We extend the MMU by the `private/shared` page detection (PSPD) logic and add a PMU for counting `private/shared` classification events.

Private/Shared Detection and Handling at the Hardware Level: Using the spare upper 4 bits of a PTE (cf. Fig. 2) we code the initialization status of a page (init bit), its `private/shared` status (P/S bit), and which processing core has initialized the page first (two "keeper" bits for core 0 to core 3). With this information the PSPD logic decides for every memory access whether page initialization, re-classification, or page access counting needs to be triggered. If the *init* bit is not set, the page was allocated by the OS due to a TLB miss handled by the page fault handler and its `private/shared` status has not been initialized yet. In this case, the PSPD logic signals an *initialization* interrupt (cf. ❶ in Fig. 2) to the OS and program execution is halted. The corresponding interrupt handler initializes tracking for this PTE and sets the *init* bit to `'1'` before resuming the execution.

If the *init* bit is set, the PSPD logic checks the *P/S* bit next. A set *P/S* bit indicates an access to a shared page and the PSPD logic generates an *shared access* counting signal for the PMU. If *P/S* bit is `zero`, i.e., the page is marked private, the PSPD logic checks whether the ID of the interrupted core is identical to the core ID in the requested PTE. If the IDs are identical, the PSPD logic generates an *private access* counting signal for the PMU. Otherwise a private page is going to be accessed by an additional core and a re-classification of the page is required. In this case the PSPD logic signals a *private-to-shared change* interrupt ❷ and the OS updates the P/S bit of all PTEs pointing to the addressed physical page as `shared`.

To avoid inaccuracies, the PSPD does not generate counting signals for memory accesses in the OS kernel mode. This is done by checking whether the 4-bit value of the Address Space Identifier (ASI) of the current access equals `0xA` [14].

Extending the PMU: The original PMU in Fig. 2 has been extended by five new event counters that are connected to the PSPD-logic (see Tab. I) [5]. The benefit of the PMU is that it integrates into the standard Linux performance measurement infrastructure seamlessly. All features of the `perf` system can be used with the new event sources without modifying any Linux kernel code.

Table I: New events implemented by the LEON3 PMU.

Event ID	Number of
`0x10`	private accesses
`0x11`	shared accesses
`0x12`	total accesses
`0x13`	page classification initializations
`0x14`	private-to-shared page reclassifications

B. OS Integration

Fig. 3 shows how hardware and software memory management mechanisms work and interact. Assuming a page being accessed for the first time, the TLB will not have yet cached the according virtual-to-physical address translation and returns a miss (cf. (1) in Fig. 3). This triggers a page table walk that will not find an entry in the page table of the process. The MMU then raises an exception causing the page fault handler of the OS allocating a physical page and a new PTE as well as clearing the PTE's *init* bit (2). The restarted address translation process causes a TLB miss again but we will find a valid PTE this time and cache the according address translation in the TLB. The address translation is restarted again, finding the desired memory address translation in the TLB and activating the PSPD logic (3).

Initialization Handler: If the PSPD logic detects that a user space page has being accessed and not initialized for `private/shared` classification yet, it raises a trap condition invoking the OS initialization handler (4). The inner logic of the handler is shown in Fig. 4. The handler first checks whether other processes are already using the according physical page. To avoid traversing all PTEs, Linux' reverse

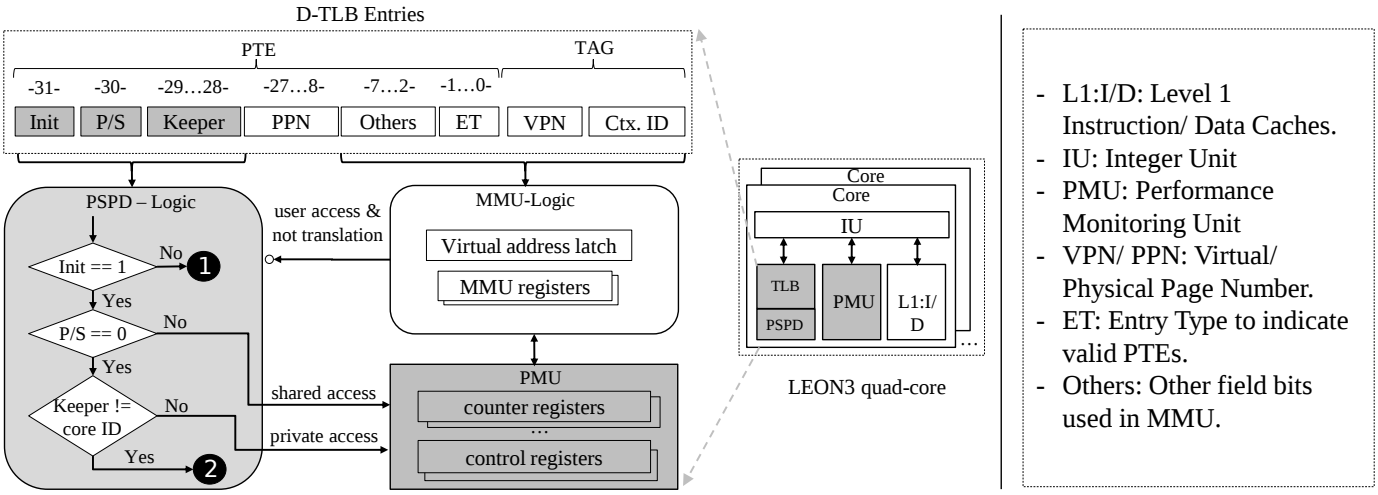


Figure 2: Private Shared Page Detection (PSPD) logic. Gray-colored elements extend the original LEON3 processor.

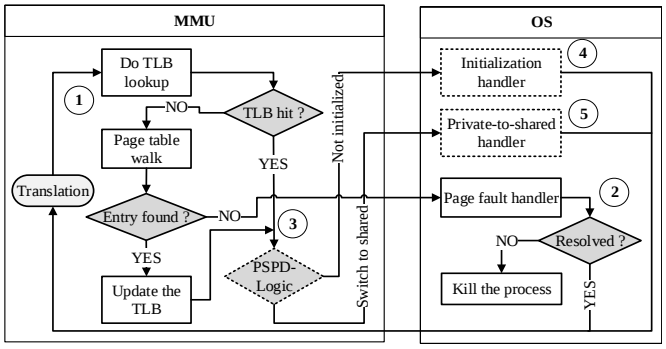


Figure 3: Flowchart of the detection mechanism.

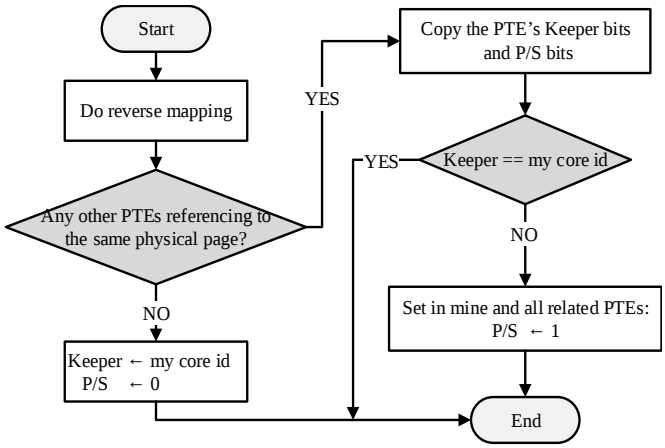


Figure 4: Inner logic of the PSPD initialization handler.

mapping is used. In case no other process is using this physical page, the page is marked private (P/S bit ← 0) and its "keeper" bits are set to the interrupted core ID. Otherwise the "keeper" and P/S bits are copied to the new PTE. Should the "keeper" bits be different from the interrupted core ID, i.e., the current core is accessing a page managed by another core, the P/S bits of all related PTEs are set to `shared` (P/S bit ← 1) and the according cache lines in the TLBs are flushed using the *Selective TLB Flush* mechanism of the SPARC architecture. Finally, the "init" bit is set to done (init bit ← 1) and the halted program is resumed.

Private-to-shared Handler: Whenever i) more than one core wants to share the same physical page, ii) a process is migrated to a different core, and iii) a child process is forked onto a different core[1], the PSPD logic detects during the first access to the previously as `private` marked page that the "keeper" bits are different from the current core ID. That means that the current core is not the core that initialized the page and consequently the page has to be re-classified as `shared`. The PSPD logic interrupts the *private-to-shared handler* (5), which uses reverse mapping to set the correct shared status in all related PTEs and to flush the according TLB cache lines of all cores.

[1]All PTEs of the parent process are copied to the child process by `fork()`.

V. Evaluation and Results

In this section we introduce the configuration of the experimental system and the experiment methodology, show results on `private/shared` memory access classification, and measure and discuss the computational overhead.

A. Evaluation Methodology

For our experiments we have selected a representative set of programs from different application domains: sequential applications from Mibench [15] and parallel applications from ParaMibench [16], PARSEC [17], and SPLASH2 [18] benchmark suites. Tab. II summarizes our selection.

Table II: Benchmarks used for evaluation.

Benchmarks	Applications
Mibench	Blowfish-Dec./-Enc., FFT, JPEG-Dec/-Enc., Quicksort.
ParMibench	Patricia.
PARSEC	Blackscholes, FLuidaminate.
SPLASH2	Cholesky, FMM, Raytrace, Water-nsquared/-spatial.

The configuration of the LEON3 processor is presented in Tab. III. As profiling applications on a real world system is subject to small deviations due to random effects of hardware components and OS scheduling as well as memory allocation strategies, we have repeated each experiment 32 times.

Table III: System configuration of the LEON3 quad-core on Xilinx ML605 board.

Parameters	Configuration
Clock Frequency	50MHz
Floating Point	Software
Memory	1GB DRAM
I/D-TLB	split, 8 entries
L1:I & L1:D	4KB, 8KB - direct mapping {16,32}-bytes/line, Snooping Protocol
Linux Kernel	2.6.36.4 patch from Gaisler
Compiler	Pre-built Linux, toolchain from Gaisler
PMU	8 event counters

B. Private/Shared Classification

The classification results for sequential and parallel applications are shown in Fig. 5. The plots report for each benchmark the percentage of accesses to private and shared pages w.r.t. the overall number of accesses as well as the standard deviation for 32 runs. Fig. 5 (a) summarizes the results for sequential applications with a share of private accesses between 93% and 97%. On average, the detected private accesses amount for approximately 95% over all evaluated applications. Although we have single threaded programs we observe 5% shared accesses. This comes from the OS scheduler that is free to migrate processes to any of the four available cores. On migration, private pages become shared in our implementation. When pinpointing each of these applications to a dedicated core using the `affinity` feature of Linux we experience 100% private accesses. Fig. 5 (b) presents the results for the parallel applications. As expected, they show a higher fraction of shared page accesses with 3% to 40%, where `blackschole`, `cholesky`, and `fmm` show the most intensive exchange of data between processes. On average, 83% of accesses are private and 17% are shared.

When comparing our results to the work of Cuesta et al. [2], we observe a notable discrepancy in the average number of accesses to private pages for page-based `private/shared` classification of parallel benchmarks (Cuesta et al.: 57%, our work: 83%), which can be explained by the different processor and cache architectures and a different selection of benchmarks. Much closer to our results are the numbers of Hardavellas et al. [3]. About 75% of memory accesses are targeting private pages in their work. An important difference is that Hardavellas et al. used fewer but much larger data base benchmarks. Additionally, the authors have a slightly extended `private/shared` classification scheme where `private` pages of a migrated process may stay `private`.

C. Overhead

Based on our LEON3 implementation, we report on the overhead induced by the `private/shared` handling. The computational overhead splits mainly into additional kernel time spent in two interrupt handlers and into a higher number of TLB misses due to re-classification of private pages. Tab. IV reports for all benchmarks the time spent in user and kernel code with and without the PSPD mechanism activated (columns "Application Time" and "Kernel Time") as well as the relative increase of kernel time. Since in this work we implemented the `private/shared` classification but did not selectively control the coherency mechanism, our expectation was that mainly the kernel time will increase. This is confirmed by the numbers in the Tab. IV where application times with and without PSPD differ insignificantly while kernel times increase by 20% to 85% when enabling PSPD. The last column of Tab. IV shows that the percentage of additional TLB misses due to re-classification of private pages as `shared` also stays rather low. An exception is the `patricia` benchmark that spends 37% of its runtime kernel mode, which increases to 45% using our PSPD logic.

Table IV: Overheads induced by `private/shared` classification. Time is reported in seconds.

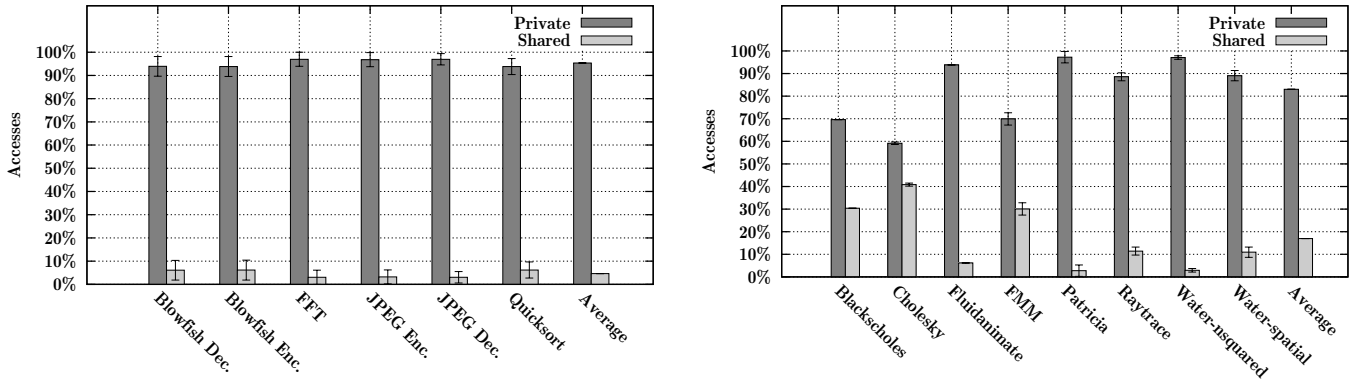
	Application Time [s]		Kernel Time [s]		Kernel Time[%]		
Sequential	No PSPD	With PSPD	No PSPD	With PSPD	No PSPD	With PSPD	Δ TLB misses
Blowfish Dec.	7.04	7.00	0.23	0.23	3.16%	3.24%	1.13%
Blowfish Enc.	7.02	6.98	0.24	0.25	3.31%	3.42%	0.74%
FFT	7.93	7.88	0.04	0.05	0.50%	0.69%	0.03%
JPEG Enc.	2.94	2.93	0.15	0.19	4.85%	6.00%	1.33%
JPEG Dec.	1.00	1.09	0.17	0.21	13.39%	16.51%	2.05%
Quicksort	1.87	1.84	0.22	0.29	10.53%	13.52%	0.13%
Parallel							
Blackscholes	2.85	2.82	0.09	0.12	3.06%	4.08%	1.57%
Cholesky	91.99	91.23	0.71	1.12	0.77%	1.21%	4.09%
Fluidanimate	41.07	40.32	1.96	3.72	4.55%	8.45%	2.29%
FMM	4.27	4.22	0.24	0.30	5.32%	6.64%	0.32%
Patricia	9.57	9.24	5.66	7.64	37.16%	45.27%	12.11%
Raytrace	406.75	413.27	0.79	1.48	0.19%	0.36%	0.12%
Water-nsquared	1.60	1.62	0.14	0.18	8.05%	10.00%	2.50%
Water-spatial	73.12	74.71	0.15	0.20	0.20%	0.27%	0.12%

Our overhead analysis shows that the overhead of a simple page-based `private/shared` detection approach strongly depends on the benchmark. While in the case of the `patricia` benchmark the overhead is notable, for many applications the time spent in user mode is dominant which makes the overhead induced by PSPD insignificant.

VI. Conclusion and Future Work

In this paper we have presented a run-time `private/shared` classification of memory accesses using an FPGA-based LEON3 multi-core processor and a regular Linux 2.6 kernel. Experiments with benchmarks from the Mibench, ParaMibench, PARSEC, and SPLASH2 application suites provide insights into the numbers and ratios of private and shared page accesses. Compared to other, simulation-based approaches our prototype is to the best of our knowledge the first implementation of a real prototype which enables us to i) provide page classification data with much higher precision and ii) quantitatively evaluate the induced overhead.

While our current implementation is limited to four cores by taking advantage of the unused 4 bits available in the PTE,



(a) Sequential applications

(b) Parallel applications

Figure 5: Results of the `private/shared` classification by our PSPD platform for sequential and parallel applications.

a larger number of cores can be supported with low overhead by enlarging PTEs with additional bits.

Besides possible optimizations with respect to the re-classification of pages as private, future work will mainly concentrate on levering our runtime page classification approach for improving multi-core architectures. For example, switching off unnecessary cache coherency mechanism activations can help save substantial amounts of energy, reduce the load on memory busses, and possibly speed up applications on multi-core processors. Our mechanism could also be used in systems with a small conventional cache for shared blocks and a larger cache without a coherency mechanism for private data blocks. Moreover, `private/shared` detection may offer an opportunity for the optimization and simplification of synonym detection mechanisms with the prospect of allowing virtual cache designs become less complex.

VII. Acknowledgement

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

References

[1] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010, pp. 111–122.

[2] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2011, pp. 93–104.

[3] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2009, pp. 184–195.

[4] A. Esteve, A. Ros, M. E. Gmez, A. Robles, and J. Duato, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 748–761, 2016.

[5] N. Ho, P. Kaufmann, and M. Platzner, "A hardware/software infrastructure for performance monitoring on leon3 multicore platforms," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[6] J. J. Valls, A. Ros, J. Sahuquillo, and M. E. Gomez, "Ps-cache: an energy-efficient cache design for chip multiprocessors," *The Journal of Supercomputing*, vol. 71, no. 1, pp. 67–86, 2015.

[7] Y. Li, R. Melhem, and A. K. Jones, "A practical data classification framework for scalable and high performance chip-multiprocessors," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 2905–2918, 2014.

[8] J. M. Cebrin, A. Ros, R. Fernndez-Pascual, and M. E. Acacio, "Early experiences with separate caches for private and shared data," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*, 2015, pp. 572–579.

[9] Y. Li, R. Melhem, and A. K. Jones, "Ps-tlb: Leveraging page classification information for fast, scalable and efficient translation for future cmps," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 28:1–28:21, Jan. 2013.

[10] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2012, pp. 241–252.

[11] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb: A token-based page classification approach," in *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. ACM, 2016, pp. 26:1–26:13.

[12] H. Hossain, S. Dwarkadas, and M. C. Huang, "Pops: Coherence protocol optimization for both private and shared data," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2011, pp. 45–55.

[13] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian, "Swel: Hardware cache coherence protocols to map shared data onto shared caches," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2010, pp. 465–476.

[14] Aeroflex Gaisler, *Grlib*. [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf

[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 3–14.

[16] S. M. Z. Iqbal, Y. Liang, and H. Grahn, "Parmibench - an open-source benchmark for embedded multiprocessor systems," *IEEE Computer Architecture Letters*, vol. 9, no. 2, pp. 45–48, Jul. 2010.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008, pp. 72–81.

[18] 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 (ISCA)*. ACM, 1995, pp. 24–36.