Evaluating Private vs. Shared Last-Level Caches for Energy Efficiency in Asymmetric Multi-Cores

Anthony Gutierrez
Adv. Computer Architecture Lab.
University of Michigan
EECS Dept.
Ann Arbor, MI, USA
atgutier@umich.edu

Ronald G. Dreslinski
Adv. Computer Architecture Lab.
University of Michigan
EECS Dept.
Ann Arbor, MI, USA
rdreslin@umich.edu

Trevor Mudge
Adv. Computer Architecture Lab.
University of Michigan
EECS Dept.
Ann Arbor, MI, USA
tnm@umich.edu

***Abstract*—In this work we explore the tradeoffs between energy and performance for several last-level cache configurations in an asymmetric multi-core system. We show that for switching threads between cores at intervals on the order of 100k or more instructions, the performance difference is negligible when private last-level caches are used in place of shared last-level caches. Thus, last-level caches can be matched to meet the needs of their host core in order to improve energy efficiency. In particular, we show that when private last-level caches are used to maintain thread state, in conjunction with energy-saving optimizations, the energy delay product of the last-level caches can be reduced by 25% on average for switching frequencies on the order of an operating system scheduling quanta—e.g., every 1 million instructions. Further, the optimizations we propose, such as power-state-aware data forwarding, are simple to implement, and the necessary support for them is already present in most current architectures.**

I. Introduction

Moore's Law coupled with Dennard Scaling had allowed the microprocessor industry to experience exponential performance gains for nearly 40 years. The end of Dennard Scaling has lead to the phenomenon known as *dark silicon* [7, 24], where not all portions of the chip may be active at once without exceeding thermal and power restrictions. In response, there have been several proposals that use specialized, asymmetric cores to operate in the presence of these restrictions [8, 9]. In these proposals, each core is designed to be suited for a particular application or phase of an application. Because only a fraction of the cores are active there is a need to support thread switching and associated working set migration.

A commercial example of a dark silicon asymmetric design is ARM's big.LITTLE system [10]—a single-ISA, asymmetric multi-core system consisting of one out-of-order core (big) and one in-order core (LITTLE). In such systems threads are switched between the cores so that only one of the two cores is active at a time. An increase in energy efficiency can be achieved by matching the thread's computational requirements to the processor that best fits its needs. When a switch occurs, the thread's working set is migrated from one core to the other. In a system where switching is common, preserving a thread's working set in the cache is critical—at the time of a switch, the inbound core's caches are typically cold with respect to the newly running thread. This cold start causes the thread to experience many cache misses, leading to long-latency memory accesses. One approach to overcoming this limitation is to allow the thread to run for a sufficiently long time to amortize the cache warmup time. However, if frequent switching is desired, or even required, this approach may not be sufficient.

Because minimizing post-migration cache warmup time is critical to preserve performance, any system with frequent switching of threads between cores must provide some mechanism to migrate the thread's working set efficiently. A significant portion of the working set is usually contained in the last-level cache(s) (LLC), allowing them to be used as a means to support working set migration. When using LLCs to support this migration there are two choices: 1) a shared LLC or 2) private LLCs. With a shared LLC a larger portion of the thread's working set can be kept in a relatively fast level of memory. However, a shared LLC is often over provisioned to suit the needs of the big core, and must always be powered on, reducing

This work was supported by ARM.

978-1-4799-3770-7/14/$31.00 ©2014 IEEE

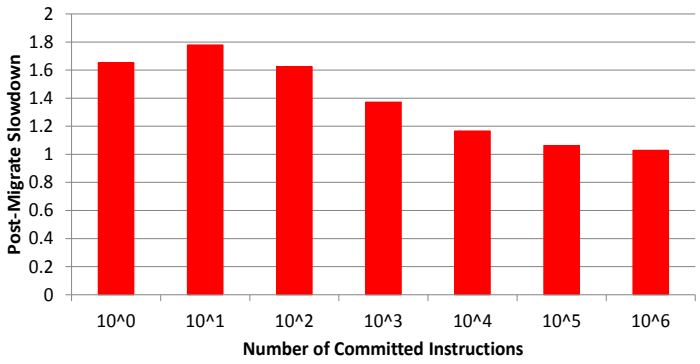


Fig. 1: Slowdown of private caches relative to a shared cache. The time to commit various numbers of instructions after a switch when using private last-level caches (normalized to a system with a shared last-level cache). The working set migration overheads are amortized after 100k instructions and the performance difference between shared and private become negligible.

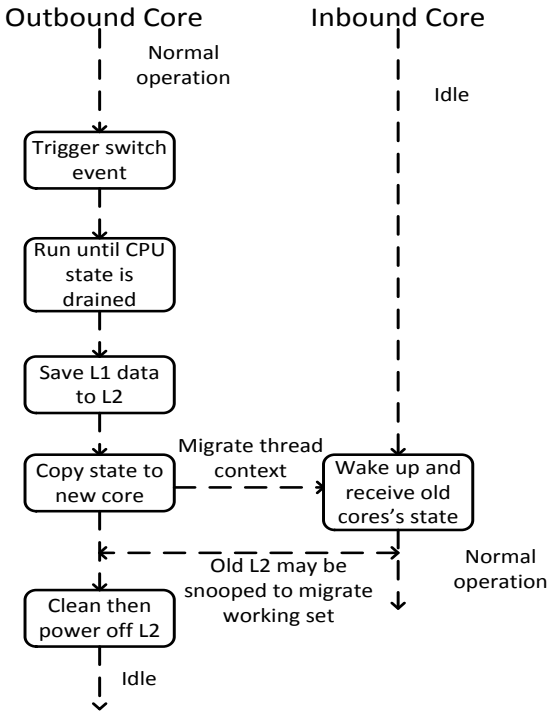


Fig. 2: Switching flow in our asymmetric system.

energy efficiency. On the other hand, in a system where private LLCs are used, each cache may be designed to suit the needs of its host core. To aid in faster working set migration, the outbound core's cache may be left powered on to satisfy coherence requests and avoid main-memory accesses. The cache may be powered off once the working set has migrated from the outbound core. However, using private LLCs incurs some overhead. One source of this is the additional LLC coherence misses that would otherwise be satisfied by a shared LLC.

In this work we explore LLC organizations to support faster working set migration and improve energy efficiency. The scope of this work focuses on *how* to support fast switching of threads between cores. The associated problem of determining *when* to switch is the subject of several recent publications—see prior work in Section VI. Results show that the number of instructions between switches can be as little as 100k instructions before private LLCs show significant performance degradation compared to shared LLC configurations.

In addition we explore private LLC management policies that reduce the portion of the working set that must be written back to DRAM, and enable cache power gating once the data has migrated. We reduce the number of LLC writebacks when the LLC's host core is powered off, and accelerate working set migration, by modifying the coherence protocol to enable ownership forwarding on read snoops when the LLC cache is powered off. Results show we are able to improve energy efficiency with a private LLC when switching every 1 million instructions by 25% on average over a shared LLC.

The rest of this paper is organized as follows: in section II we describe the performance impact of using private vs. shared LLCs. In section III we describe the baseline architecture as well as our optimized private LLC cache system. Section IV details our simulation framework and workloads. A discussion of the results of our investigation is presented in section V. We give an overview of related work in VI, and concluding remarks in section VII.

II. Last-Level Cache Organization

In this section we describe the different types of LLC organizations implemented in modern asymmetric multi-core architectures, as well as the performance tradeoffs for each. We show that using private LLCs does not have a significant impact on performance when compared to a shared LLC and conclude that the use of private LLCs to support working set migration is effective for switching as often as every 100k instructions.

Big Cache Parameters	Value
Cache Line Size	64B
L1 Size	32kB
L2 Associativity	16-Way
L2 Size	1, 2, 4MB
Little Cache Parameters	**Value**
Cache Line Size	64B
L1 Size	16kB
L2 Associativity	16-Way
L2 Size	128, 256, 512kB
Shared L2 Parameters	**Value**
Cache Line Size	64B
L1 Size	32kB
L2 Associativity	16-Way
L2 Size	128, 256, 512kB, 1, 2, 4MB

TABLE I: Hardware parameters. The cache system parameters used in our experiments. A sweep of various different last-level cache sizes were explored for both shared and private L2 configurations. To isolate the effects of cache asymmetry we use a simple 1 CPI CPU for both the big and little systems.

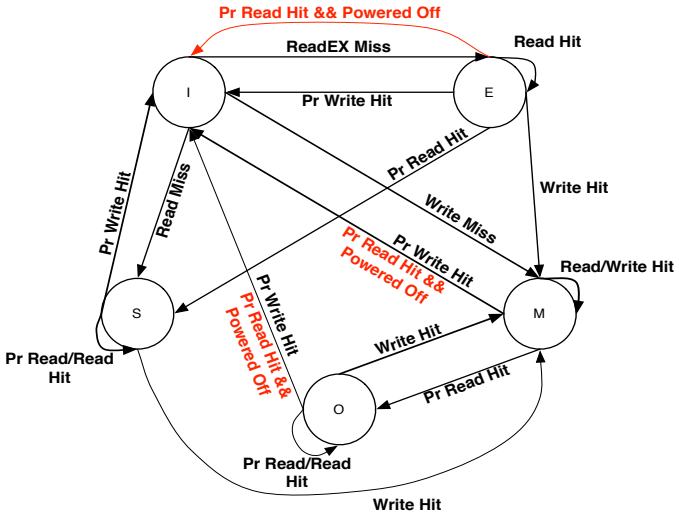


Fig. 3: Modified MOESI coherence protocol with power-state aware exclusivity forwarding. The transitions from the O and M states to the I state have been modified to also transition on read snoops when the cache owning the line is powered off. The transition from the E state to the I state has been added for the case when the cache owning the line snoops a read.

A. Preserving Thread Working Sets in Last-Level Caches

Preserving thread working sets is critical for mitigating the performance impact of switching threads between cores, particularly when switching occurs frequently. Prior works have developed techniques that can preserve cache state during switches [4, 14, 15, 23]. These techniques can significantly improve performance in systems that switch frequently, however, the techniques they employ can increase energy consumption and require complex hardware or software changes. In this work we explore the use of LLCs to maintain cache state across switches, thus mitigating the performance penalty of working set migration while at the same time reducing energy consumption.

B. Shared Last-Level Cache

Because LLC misses incur a very high penalty, it may be beneficial to utilize a shared LLC when threads experience frequent switches between cores. This ensures that important LLC state is preserved across switches. However, in an asymmetric multi-core system the LLC is typically over-provisioned to support the needs of the big core—parameters such as cache size, associativity, number of MSHRs, number of read and write ports, block size, etc., are often higher than they need to be when the little core is active.

C. Private Last-Level Caches

Private LLCs offer a lower power solution for asymmetric multi-core systems; the LLCs can be tailored to suit the needs of their host core, and may even be powered off when they are inactive. However, a balance must be struck between the amount of cache state that is preserved and how much power is consumed. If we completely power off a LLC when a switch occurs, performance can be significantly reduced. If the private LLCs are always on, even when its host core is powered off, then the power consumption will likely be worse than using a single large LLC.

D. Performance Impact of Using Private Last-Level Caches

Brown et al. model a system, with private LLCs, where thread migration occurs every 1 million instructions [4]. They show that the post-migrate slowdown,

in terms of the time it takes to commit 1, 10, 100, ..., 10^6 instructions, is insignificant after around 10 thousand instructions are committed. However, at shorter switch intervals the performance impact is more acute, as much as 8× worse in some cases. The slowdown reported by Brown et al. is relative to a system where all cache state is instantly migrated along with the thread, as the focus of their work was not on the memory system. Thus, their results do not provide much insight into what, if any, performance loss is incurred in a system that uses private vs. shared LLCs.

We repeat their experiment on a system with a shared LLC and on a system with private LLCs. The shared LLC is 1MB and in the private LLC case we use one 1MB cache and one 256kB cache. We run a memory-intensive subset of the SPEC CPU 2006 benchmarks on a two core system where only one core is active at any given time, and a switch is forced every 1 million instructions. Results shown in Figure 1 indicate a negligible performance impact caused by using private vs. shared LLC(s) for coarse grained switching intervals ($\geq$100k instructions). However, the time to commit 1 to 10,000 instructions is delayed between 20-60%. The conclusion from this data is that private LLC designs are still viable options for switching intervals down to 100k instructions, significantly shorter than the OS schedulers time quanta. Beyond this point ($<$100k instructions) shared LLC designs are necessary. As noted, our goal is to explore what performance/energy tradeoffs can be made when designing an asymmetric multi-core system and how a system should look if frequent switching is desired.

III. Private Last-Level Caches for Energy Efficiency

In this section we describe the baseline architecture and our enhanced design. We detail each of the LLC optimizations and how they are implemented in our design.

A. Baseline Architecture

Our baseline architecture consists of a two core asymmetric multiprocessor with per-core split instruction and data caches, and a shared LLC. In this work we focus on the asymmetry in the memory systems, thus we use a simple 1 CPI CPU model for both the big and the little CPUs. The memory system parameters are listed in table I. We implement the same switching infrastructure in this baseline system as we do in our system.

B. Switching Policy

A switch is triggered, by the hardware, every N committed instructions, where N is swept across a range switching intervals. At the time of a switch, execution is halted and all CPU thread context is transferred from the outbound core to the inbound core. Once a thread has switched away from a core, the core is powered off and does not execute any instructions until the thread switches back to it. In the cycles after the switch is triggered, the outbound core's L1 instruction and data caches are cleaned and then powered off. The basic operation of our system is shown in Figure 2. We use a constant instruction interval to trigger the switch because the goal of our work is to determine how to design memory systems to support fast core switching. This is orthogonal to the question of when hardware or software should trigger a switch. As noted, several related works on heuristics to determine when to switch, and which core to switch to are provided in Section VI.

C. Cache Asymmetry and Power Down Enhancements

With the use of private LLCs it becomes possible to utilize LLCs that are suited to the needs of their host core and to power them off when they are no longer required to be powered on. In our system, when a core is powered off its LLC is left powered on so that any dirty data it contains may be snooped by the active core; this will help reduce the number of unnecessary long-latency DRAM accesses by satisfying the requests with a cache-to-cache transfer. After a certain period of time the likelihood that a dirty cache will be read is low, thus all dirty lines in the LLC are cleaned so that the entire cache may be safely powered off. As we will see in V the overall energy consumption for the LLCs in a private LLC configuration can be significantly reduced when compared to a shared LLC.

D. Power-State-Aware Ownership Forwarding Enhancements

To reduce the number of dirty lines that must be written back from the LLC when it is powered off, we add another state transition to the standard cache coherence protocol. Figure 3 shows the state diagram for our modified MOESI protocol and figure 4 shows our asymmetric multi-core system, illustrating ownership forwarding. With power-state aware ownership forwarding, any time a line in the *owned*, *exclusive*, or *modified* state snoops a read request from another LLC, ownership of the line is forwarded along with the data to the reader.

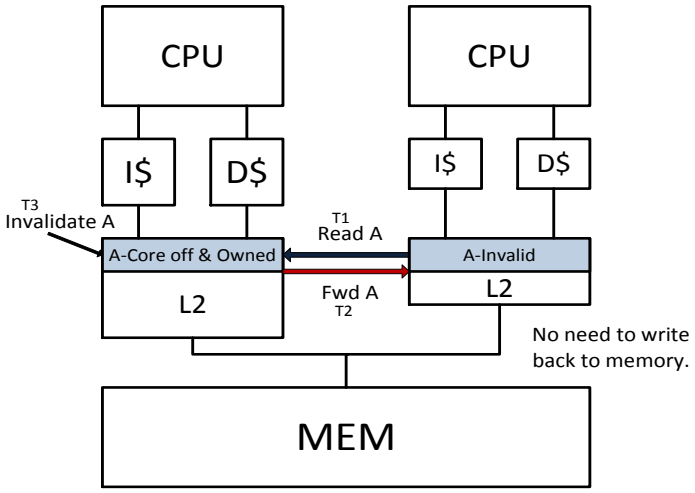


Fig. 4: Hardware configuration utilizing private last-level caches. The private last-level configuration employs power-state aware ownership forwarding, and may be powered down once a thread has warmed up its caches after a migration. In this example, at time T1 a read is snooped by the L2 on the left while its host core is powered off, at T2 it forwards ownership of line *A* along with the data to the snooping core, finally at T3 it invalidates its own copy of line *A* ultimately preventing a DRAM writeback of that line when the L2 is powered off.

Because a LLC that is powered down cannot service a read request from its host CPU it is not necessary to preserve ownership in a LLC whose host is powered off.

As will be shown in section V, power-state aware ownership forwarding does not provide much benefit when switching is performed infrequently, however when threads are migrated very frequently it can improve the energy delay product by around 15% on average when compared to a system without any optimization for some of the benchmarks. The energy savings provided by power-state aware ownership forwarding come at virtually no cost—many current systems provide mechanisms to forward ownership on a read snoop if it may provide some benefit, e.g., ARM's ACE protocol is one such system [1].

IV. Experimental Methodology

A. Benchmarks

For all experiments a memory-intensive subset [11, 12] of the SPEC CPU 2006 benchmarks is used. The benchmarks we use are listed in table II. Each benchmark is run for 1 billion instructions using the *ref* input set. Functional simulation is used to fast-forward through kernel boot.

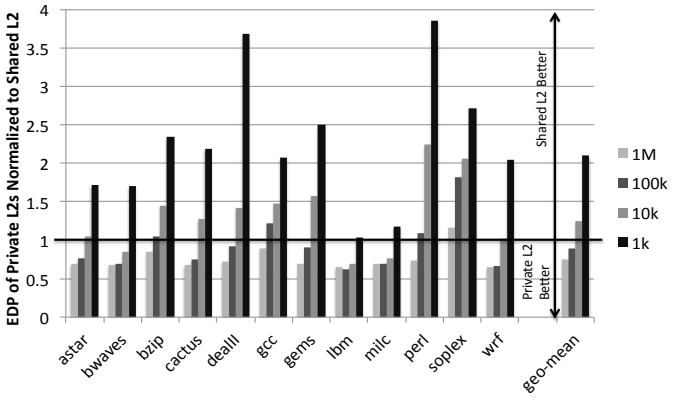
Benchmark	Memory Footprint
astar	313 MB
bwaves	881 MB
bzip2	856 MB
cactusADM	879 MB
dealII	564 MB
gcc	932 MB
GemsFDTD	838 MB
lbm	416 MB
milc	676 MB
perlbench	580 MB
soplex	457 MB
wrf	701 MB

TABLE II: SPEC CPU2006 benchmarks used in this study. All benchmarks were run using the *ref* input set.

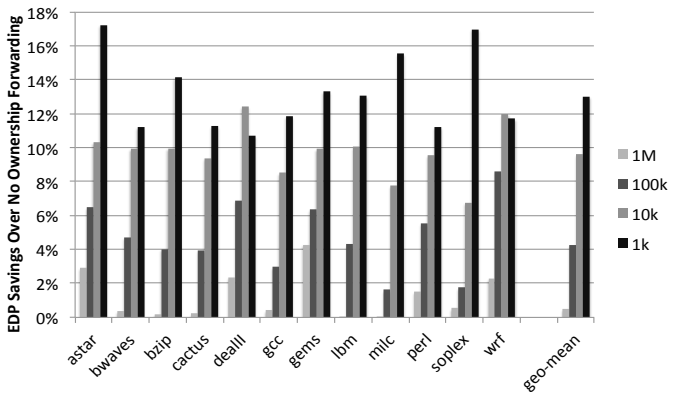
B. Simulation Infrastructure

A modified version of the gem5 simulator [3] is used for all experiments. We model our thread-migration policy after the big.LITTLE asymmetric multi-core CPU from ARM [10]. In our setup only one core is ever active at a time and switching happens at a very coarse granularity—on the order of operating system scheduling quanta. However, we also explore the effects of switching more frequently—on the order of 1,000 instructions. To model the context migration, i.e., the copying of all thread state from core-to-core, as well as to power down a core, we build off of gem5's *drain* and *switchout* functionality. After *N* instructions have committed we change the currently running CPU to a *draining* state. While in the draining state a CPU does not fetch new instructions, it simply continues to execute until all of the state drains out of its internal structures, e.g., reorder buffer, load-store queues, etc.

In the case of private LLCs we add new *owned_off* and *modified_off* state for each cache block. This state is similar to the standard O state of the MOESI protocol except that it contains information regarding the state of the cache's host core, i.e., whether or not it is powered on. To implement power-state aware ownership forwarding, we make a change to the cache snooping functionality—if a cache block in the *owned_off* or



(a) Energy delay product for SPEC over various switching frequencies relative to a shared last-level cache.



(b) Energy delay product with ownership forwarding relative to caches with no ownership forwarding.

Fig. 5: EDP. Figure 5a shows that for switching intervals as low as 100k, private LLCs are more efficient. For switching intervals of 1M, EDP is 25% lower on average for optimized private LLCs. Figure 5b shows that for switching intervals where private LLCs are better (100k, 1M), 4% of the EDP savings are attributed to ownership forwarding on average. We use a 1MB LLC in the case of a shared LLC. For private LLCs, the big LLC is 1MB and the little LLC is 256kB.

modified_off state snoops a read request from another core it will forward ownership along with the block's data to the other cache, then it will invalidate its own version of the block. After a certain period of time a running core will notify the switched out cache that it can clean and invalidate itself. When an L2 cache receives this notification, it iterates through every cache block and, if dirty, writes it back to memory; once this process is complete the L2 cache is powered off. Table I lists the different cache parameters we evaluate in our system. In addition to varying these cache parameters, we also evaluate several different switching intervals: every 10^n instructions $n \in 3...6$, and various times at which to clean a cache after a migration: after 10^n instructions $n \in 1...5$.

All of our power and energy numbers were obtained using CACTI [19].

V. Results

Figure 5a lists the energy-delay product for each workload relative to using a shared L2 cache. As can be seen in Figure 5a, even when switching occurs in intervals as short as 100k instructions the EDP favors private LLCs for most benchmarks. When thread migration occurs every 10k instructions, the EDP favors a shared LLC for most benchmarks. For switching intervals on the order of 1M instructions EDP is 25% lower for optimized private LLCs.

To quantify the impact of ownership forwarding we compare our optimized private L2 system with a private L2 system with no ownership forwarding; these results can be seen in figure 5b. As can be seen in this graph, ownership forwarding always gives greater energy efficiency, particularly when thread migration is very frequent. One interesting result is that the ownership forwarding optimization can help regain some of the efficiency lost at a short switching intervals. One of the reasons for this can be seen in figure 6, which shows that the DRAM accesses can be reduced around 1% when thread migration occurs every 100k instructions. Ownership forwarding allows owned lines to be forwarded from a powered-down cache on a read request, thus reducing the number of dirty lines that need to be written back when a cache is cleaned; it also also helps reduce the number last-level misses when a write request follows a read request.

VI. Related Work

In this section we describe previous research relating to asymmetric multi-core architectures. We summarize the latest research in the areas that are most crucial to asymmetric multi-core systems.

A. Asymmetric Multi-Cores to Save Power

Prior works have observed the benefit of using asymmetric multi-core architectures with switching for im-

proved efficiency; not only to improve the performance and efficiency of multi-threaded applications, but single-threaded applications and operating system code as well [2, 6, 13, 15, 17, 16, 18, 20, 23, 22]. There are even commercial products in the works that will explicitly utilize asymmetric multi-core architectures, e.g., ARM's big.LITTLE processing [10]. The common observation in all of these works is that many applications have periodic phases of execution that have differing requirements. Asymmetric multi-core architectures can be used to service the different phases of a program; improving energy efficiency without reducing performance.

B. Asymmetry Aware Scheduling

Several prior works have focused developing hardware and software (at the OS scheduler level) techniques to determine when it is most beneficial to migrate a thread, and on which core a thread should be scheduled [14, 21, 26]. These works typically focus on determining when to switch a thread, and on which core to switch, based on several different heuristics, including memory-level parallelism, instruction-level parallelism, and working set distribution. It is usually the case that overall system performance benefits most when threads that would conflict with each other are scheduled on different cores.

C. Preserving Cache State Across Switches

When a switch occurs, the inbound core's caches are typically cold with respect to the newly running thread; this causes the thread to experience many cache misses, causing long-latency memory accesses. One approach to overcoming this limitation is to allow the thread to run for a sufficiently long time to amortize the cache warmup time. However, if frequent switching is desired, this approach is not sufficient. Several works have proposed techniques to help reduce cache warmup time by preserving cache state across migration, that is, by essentially using computation spreading [5], or by using some sort of core-to-core pre-fetching during migration [4, 15, 23, 25]. These works typically employ sophisticated prediction schemes, either in the hardware or the compiler, to forward cache state from one core to another after a switch takes place. While these works provide significant improvements to performance, they do not fully consider energy consumption. Their schemes may require additional energy that may reduce the energy efficiency gained by using asymmetric multi-core systems.

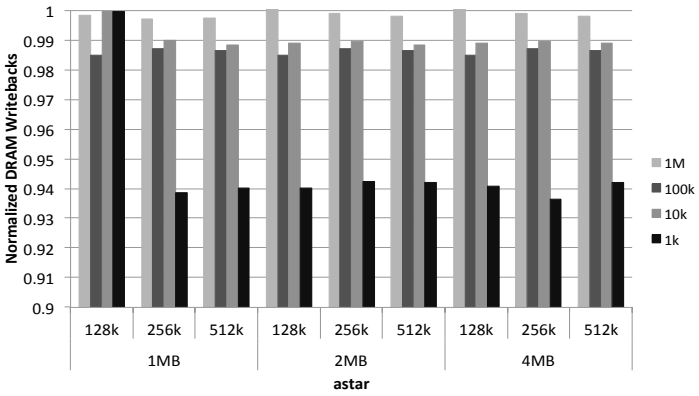


Fig. 6: Reduced DRAM accesses with ownership forwarding for the astar benchmark. With ownership forwarding the number of DRAM accesses can be reduced when compared to a system without ownership forwarding—by forwarding to the other cache the dirty lines that would otherwise need to be written back. For switching intervals that favor private last-level caches (100k, 1M) we save ∼1.5% of costly DRAM accesses.

VII. Conclusion

In this work we have studied the effects of using private vs. shared last-level caches for asymmetric multi-core systems. We have shown that utilizing private last-level caches in conjunction with energy-saving techniques, such as power-state-aware ownership forwarding, can help improve energy consumption in asymmetric multi-core systems, with little impact on performance, thus increasing overall efficiency. The last-level cache configurations and optimizations that we have implemented can improve the EDP for the last-level caches by 25% on average for the benchmarks we evaluated, when switching is on the order of an operating system scheduling quanta, e.g., 1 million instructions. Because future CMP systems will likely need to keep large portions of the chip off at any given time, computer architects will need to design systems with various types of components capable of servicing a wide range of applications, and even phases within applications. Our results suggest that systems should not only utilize asymmetric and specialized cores, but specialized memory systems as well.

Acknowledgement

The work presented in this paper was sponsored by ARM.

REFERENCES

[1] ARM, ed. *AMBA AXI and ACE Protocol Specification*. ARM. 2011.

[2] A. Bhattacharjee and M. Martonosi. "Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors". In: *the proceedings of ISCA 36*. 2009, pp. 290–301.

[3] N. Binkert et al. "The gem5 Simulator". In: *SIGARCH Compututer Architecture News* 39.2 (2011), pp. 1–7.

[4] J. Brown, L. Porter, and D. Tullsen. "Fast Thread Migration via Cache Working Set Prediction". In: *the proceedings of HPCA 17*. 2011, pp. 193–204.

[5] K. Chakraborty, P. M. Wells, and G. S. Sohi. "Computation spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly". In: *the proceedings of ASPLOS XII*. 2006, pp. 283–292.

[6] T. Constantinou et al. "Performance Implications of Single Thread Migration on a Chip Multi-Core". In: *SIGARCH Computer Architecture News* 33.4 (2005), pp. 80–91.

[7] H. Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *the proceedings of ISCA 38*. 2011, pp. 365–376.

[8] N. Goulding-Hotta et al. "GreenDroid: An architecture for the Dark Silicon Age". In: *the proceedings of ASP-DAC 17*. 30 2012-feb. 2 2012, pp. 100–105.

[9] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future". In: *Micro, IEEE* 31.2 (2011), pp. 86–95.

[10] P. Greenhalgh. *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. whitepaper. ARM, 2011.

[11] J. L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Compututer Architecture News* 34.4 (2006), pp. 1–17.

[12] J. L. Henning. "SPEC CPU2006 Memory Footprint". In: *SIGARCH Computer Architecture News* 35.1 (2007), pp. 84–89.

[13] M. Hill and M. Marty. "Amdahl's Law in the Multicore Era". In: *IEEE Computer* 41.7 (2008), pp. 33–38.

[14] X. Jiang et al. "ACCESS: Smart Scheduling for Asymmetric Cache CMPs". In: *the proceedings of HPCA 17*. 2011, pp. 527–538.

[15] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads". In: *the proceedings of the ASPLOS XVI*. 2011, pp. 393–404.

[16] R. Kumar et al. "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance". In: *the proceedings of ISCA 31*. 2004, pp. 64–75.

[17] R. Kumar et al. "Heterogeneous Chip Multiprocessors". In: *Computer* 38.11 (2005), pp. 32–38.

[18] J. Mogul et al. "Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems". In: *IEEE Micro* 28.3 (2008), pp. 26–41.

[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0". In: *the proceedings of MICRO 40*. 2007, pp. 3–14.

[20] K. K. Rangan, G.-Y. Wei, and D. Brooks. "Thread Motion: Fine-Grained Power Management for Multi-Core Systems". In: *the proceedings of ISCA 36*. 2009, pp. 302–313.

[21] R. Strong et al. "Fast Switching of Threads Between cores". In: *ACM SIGOPS Opererating Systems Review* 43.2 (2009), pp. 35–45.

[22] M. A. Suleman et al. "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures". In: *the proceedings of ASPLOS XIV*. 2009, pp. 253–264.

[23] M. A. Suleman et al. "Data Marshaling for Multi-Core Architectures". In: *the proceedings of ISCA 37*. 2010, pp. 441–450.

[24] M. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse". In: *the proceedings of DAC 49*. 2012, pp. 1131–1136.

[25] H.-W. Tseng and D. Tullsen. "Data-Triggered Threads: Eliminating Redundant Computation". In: *the proceedings of HPCA 17*. 2011, pp. 181–192.

[26] K. Van Craeynest et al. "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)". In: *the proceedings of ISCA 39*. 2012, pp. 213–224.