Taming Data Caches for Predictable Execution on GPU-based SoCs

Taming Data Caches for Predictable Execution on GPU-based SoCs

Björn Forsberg[1] Luca Benini[1, 2] Andrea Marongiu[2,1]
[1] Swiss Federal Institute of Technology Zürich [2] University of Bologna
{bjoernf, lbenini, a.marongiu}@iis.ee.ethz.ch

***Abstract*—Heterogeneous SoCs (HeSoCs) typically share a single DRAM between the CPU and GPU, making workloads susceptible to memory interference, and predictable execution troublesome. State-of-the art predictable execution models (PREM) for HeSoCs prefetch data to the GPU scratchpad memory (SPM), for computations to be insensitive to CPU-generated DRAM traffic. However, the amount of work that the small SPM sizes allow is typically insufficient to absorb CPU/GPU synchronization costs. On-chip caches are larger, and would solve this issue, but have been argued too unpredictable due to self-evictions. We show how self-eviction can be minimized in GPU caches via clever managing of prefetches, thus lowering the performance cost, while retaining timing predictability.**

I. Introduction

Heterogeneous SoCs (HeSoCs) [1], [2] have become a promising target for the deployment of computationally intensive embedded workloads in domains such as autonomous automotive and avionics systems. These devices achieve high performance-per-watt and small form factor by sharing resources, at the cost of increased contention that may affect the timing behavior of tasks. This has prevented the widespread adoption of HeSoCs in these domains, where real-time certification is required. Because of this, software resource management techniques are in high demand.

In HeSoCs the main resource under contention is the single DRAM shared by the host and accelerator, potentially introducing delays at every load/store operation [3]. Aside throttling-based techniques [4], [5], [6], one of the most widespread approaches to overcome this is the Predictable Execution Model (PREM) [7], [8], [9], [10], [11], which allows DRAM access to the CPU and GPU in mutually exclusive time windows, during which data is copied to local memory. Computation can happen on local data when DRAM access is not allowed, to minimize system idleness. The state-of-the-art in PREM on heterogeneous CPU+GPU systems [11] uses the GPU scratchpad memory (SPM) for local storage, as the software managed data placement provides a good basis for predictable execution. However, due to their small size, SPMs are prone to significant overheads due to the synchronizations inherent to the technique. Integrated GPUs already feature much larger hardware-managed caches, that would potentially remedy this problem, but unpredictable replacement policies have previously deterred any attempts at using these.

This paper shows how the hardware-managed caches of the NVIDIA Tegra SoCs [1] can be successfully used with the PREM model, to achieve predictable execution, at better performance than the SPM-based state-of-the-art.

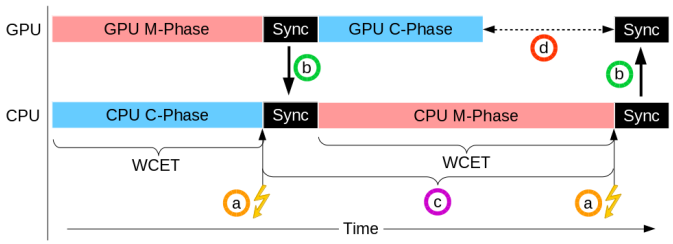

Fig. 1: The key components of a PREM interval.

II. The Predictable Execution Model (PREM)

The insight that underlies PREM is that any access that *hits* in the local memory does not depend on the shared resource, i.e., DRAM, and the worst case execution time (WCET) can not be influenced by memory contention. For misses, isolation (no impact on WCET) can be achieved by reserving the memory system exclusively for the memory access. However, as cache hit analysis is difficult for individual accesses [12], and the mechanisms required to protect them are costly, it is infeasible to do this on a per-access granularity. Instead, PREM divides the program into coarse-grained *intervals*, depicted in Figure 1, consisting of a *memory* and a *compute* phase (henceforth *M-phase* and *C-phase*). The *M-phase* is responsible for bringing all data used within the *interval* to the local (private) memory, such that the *C-phase* is guaranteed to hit in the cache. Thus, costly protection of each individual access is replaced with the protection of the coarser *M-phases*.

Heterogeneous PREM – Protecting the *M-phases* on the CPU is straight forward, as the OS scheduler has full control over which threads execute. On HeSoCs, the problem is more difficult, as the protection needs to be extended beyond the scope of the OS scheduler, to include the accelerators. This can be achieved with custom software synchronization [10], that enable memory access rights to be passed at a system-level between the two devices. These synchronizations are triggered by timer interrupts at the expiry of a WCET watchdog timer (Fig. 1 (a)), at which point the memory *token* can be exchanged between the processing units (Fig. 1 (b)). However, these synchronizations can not occur at too fine granularity: The system requires enough time to accomodate the interrupt latency and interrupt handler, as well as leaving enough time for ”useful work”. Therefore, there exists a system-dependent *minimum synchronization granularity* (MSG) (Fig. 1 (c)). In the case when the phase lengths are shorter than the MSG, the GPU kernel is forced to *idle* until the CPU becomes ready for synchronization (Fig. 1 (d)). When designing the system

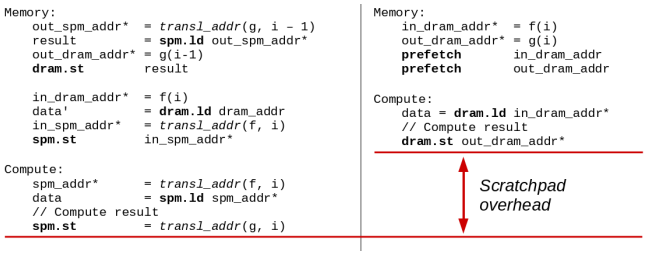

Fig. 2: SPM data movement code (left) requires more instructions than caches (right).

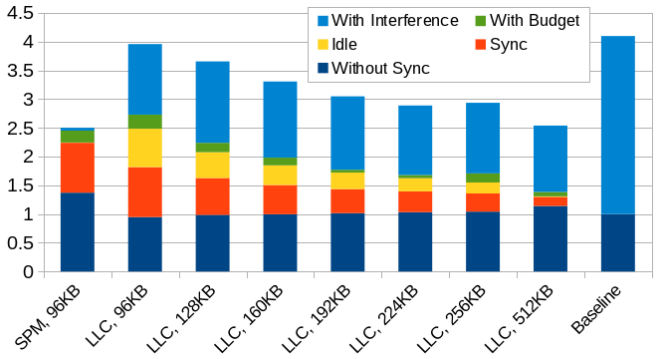

Fig. 3: The breakdown of the execution time for the *bicg-100* kernel on the SPM, LLC, and without PREM (baseline).

to account for the MSG and the length of the PREM phases, we say that we place a *budget* on the system, which affects when the watchdog timer expires.

III. The Cases for and Against Caches

This section explores how SPM and caches impact PREM.

Synchronization – On the GPU, the SPM has been successfully used as local storage for PREM intervals [11]. However, the small size of the SPM implies short PREM phases, even below the MSG, which causes the synchronization/idling overhead to blow up. Intuitively, these overheads can be overcome by increasing the granularity of the *intervals*, such that the synchronization makes up a smaller proportion of the overall execution time. On current generation heterogeneous SoCs, the last level cache (LLC) of the accelerator is much larger than the SPM ($5\times$ on the NVIDIA TX1) Thus, the use of caches promises a more effective use of the kernel execution time.

Code performance – As SPMs are software managed and explicitly addressed, they require a significant addition of instructions to manage the data allocation and data movement. This implies an overhead compared to the use of implicitly addressed caches. A simple example provided in Figure 2 highlights the difference between the two cases. Note that depending on the complexity of the address calculation, the added instructions from `transl_addr` (which transforms a DRAM address to its SPM counterpart) can be significant. In contrast, the only instructions needed for hardware-managed caches is a prefetch of the original address in the *M-phase*. Because of this, hardware caches promise additional performance benefits due to hardware-managed data placement.

Self-eviction – To ensure predictability, no cache miss must occur in the *C-phase*. To measure this we use the *compute phase miss rate* (CPMR), defined as the ratio of cache misses in the *C-phase* over the total amount of cache misses.

Thus, the key design goal for PREM is to minimize the CPMR, which makes a strong case for the SPM: The software managed data movement ensures that data brought into the SPM is guaranteed to *survive* until explicitly *evicted*. In contrast, data in the caches are managed by a fixed *replacement policy*, that selects which data to *evict* when new data is requested. If the *least-recently-used* (LRU) [12] replacement policy is used, this represents no problem. However, LRU is seldom used in commercial systems, because of the complex hardware needed to implement it. Instead, cheaper but less predictable replacement policies are used, that can lead to (more or less) random *self-evictions* of alive data from the cache. This is also the case in the NVIDIA Tegra GPUs [13].

A. Real-world example

In summmary, the cache promises smaller overheads, but the problem of self-eviction may compromise the key design goal of minimizing the CPMR. To see how this manifests in practice, we run the cache-friendly *bicg-100* kernel from the PolyBench-ACC [14] benchmark suite on the NVIDIA TX1. We compare the characteristics of this kernel on the SPM, on the LLC, and without PREM (baseline), under different interval sizes T, determined by the amount of data touched. The SPM is limited to $T \leq 2 \times 48KB$, while the LLC allows $T \leq 256KB$. The results are shown in Figure 3, where execution times are shown relative to the baseline.

From the bottom up, "without sync" shows the effect of either SPM data movements or prefetches, and the "idle" and "sync" parts show the two sources of synchronization overhead described in Section III. In these aspects, caches indeed do better than the SPM; the overheads are rapidly decreasing as T goes up, and the use of prefetches initially has a positive effect, as can be seen in the decrease in "without sync" between the SPM and LLC cases. As T increases beyond the cache size of $256KB$, we start seeing capacity misses.

After budgeting for the WCET and applying memory interference, the two top-most bars show a large difference between the SPM and the LLC. While PREM on the SPM is indifferent to memory interference, PREM on the LLC shows a significant slowdown under interference. Compared to the baseline, PREM on LLC can still perform better under interference, as the cache misses that occur in the *M-phase* are protected, but any cache miss in the *C-phase* is subject to the same slowdown. For the LLC, the bad performance under interference would occur only if the data prefetched in the *M-phase* is selected for eviction before its point of use in the *C-phase*, as the resulting cache misses in the *C-phase* would be subject to interference. This implies that the cache replacement policy is working against us, as the prefetch can not guarantee that the data is in the cache after the *M-phase*.

IV. The Way of the Cache

The main source of self-evictions in the NVIDIA GPU caches, as shown by [13], is the random replacement policy.

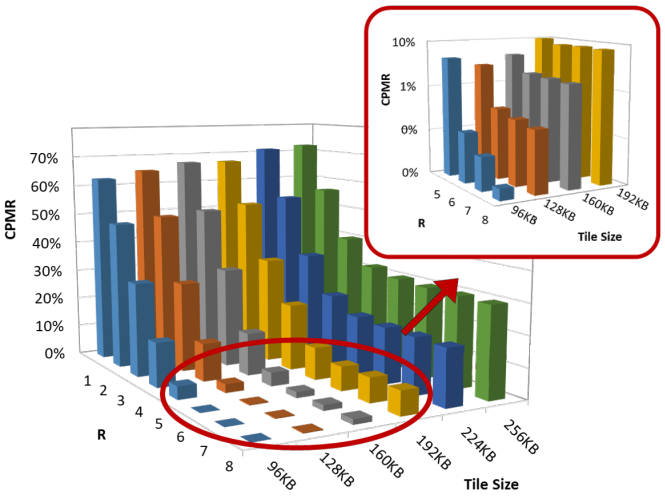

Fig. 4: The CPMR for different R and T.

Data stored in different cache ways are more or less likely to be evicted, and if the loaded data ends up in a cache way that is more likely to get evicted, it will not *survive* until the start of the *C-phase*. The authors of [13] were only able to show this effect in the L1 cache, where out of four *cache ways* per set, one was three times more likely to evict the data[1]. We refer to this way as the *bad way*, and the others as the *good ways*. To minimize the CPMR we want to have a near-zero probability that data is stored in the *bad way*. Since the eviction probabilities imply a 50% chance of data being stored to the *bad way*, we can model this as a *coin toss* (probability of getting R *heads* in a row), in which the likelyhood of using a *bad way* reaches a probability of less than 0.5% at $R \geq 8$.

We verify this intuition, and that this applies to the LLC, in Figure 4, measuring the CPMR for different R at interval sizes T. As expected, increasing the number of prefetches by a factor of R monotonically decreases the CPMR towards near-zero values. Thus, we can decrease the CPMR by choosing $R = 8$, and we refer to R as the prefetch repetition factor.

In the other dimension we explore the effects on the CPMR as the interval size T increases, and see that as T decreases the CPMR also decreases. For all $T \leq 192$ the CPMR reaches CPMR $< 10\%$, after which it increases rapidly. We know that $3/4$ of the cache ways are *good*, and $1/4$ is *bad*. That also means, that of the full cache capacity of $256KB$, only $192KB$ ($3/4$th) is available in *good* cache ways. Which in turn means:

- While $T \leq 192KB$, all data fits in the *good ways*, and with a high enough R, all data will reside there.
- Once $T > 192KB$, data must also be stored in the *bad ways*, and there is a very high risk that this data will be evicted at any cache miss. We therefore expect the CPMR to start increasing due to self-eviction.

Comparing this to the CPMR in Figure 4, we can see that this matches the observed pattern. Thus, by choosing an interval size T that is small enough to fit in the *good ways* of the cache, and repeating the prefetch operation $R = 8$ times, we

[1]In [13], the observed probabilities of evictions were $(\frac{1}{6}, \frac{1}{6}, \frac{3}{6}, \frac{1}{6})$.

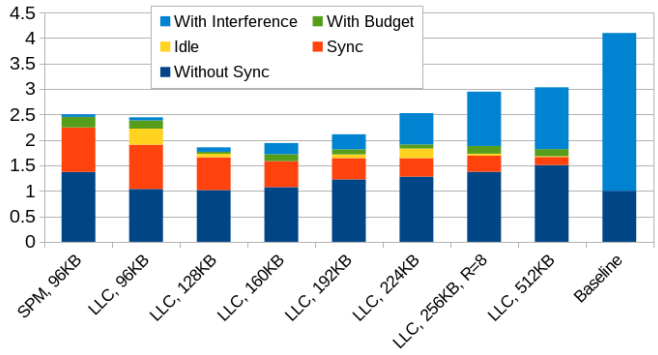

Fig. 5: The breakdown of the execution time for the *bicg-100* kernel, with a prefetch repetition R of 8.

are able to significantly reduce the CPMR, whose previously high values prevented predictable execution on the LLC.

A. Real-world example revisited

Equipped with this knowledge, we revisit the example from Section III-A, repeating each prefetch operation $R = 8$ times, and plotting the results in Figure 5. We now see, in the "without sync" part of the bars, that the code overhead slowly increases with the interval size T. To understand how this affects the execution time, lets reason about the impact of repeated prefetches. For a repeated prefetch that hits in the cache, the increase in execution time should be negligible because the low cache latency. If a repeated prefetch causes a miss, we will overcome the self-eviction by refetching it, and thus move one cache miss from the *C-phase* to the *M-phase*, expecting no change in the overall execution time. This is the case for interval sizes that fit in the *good cache ways*, when $T \leq 192KB$. However, if we repeat the prefetch, and after that *still* miss in the *C-phase*, the overall number of cache misses increases, and with it the execution time goes up. We can see that this effect starts at $T = 192KB$, where the *good* cache ways are no longer enough to hold all data of the interval.

The good news, however, is that the lowering of the CPMR has the corresponding decrease in sensitivity to memory interference ("with interference" in Figure 5), showing a positive relationship between the CPMR and predictability achievable.

V. Evaluation

This section extends the evaluation to more kernels from the PolyBench-ACC benchmark suite [14], using a subset of benchmarks for which the SPM-based PREM execution implies large overheads. We co-schedule the TX1 CPU and GPU so that both devices get an equal share of the memory bandwidth, which ensures that neither device is starved for memory. This is achieved by budgeting the *M-* and *C-phases* to equal length. The results are presented in Figure 6.

A. Optimized cache performance

In all cases, the SPM-based state-of-the-art performs significantly worse than the LLC, and we can confirm the previous results that the best configuration is to use a T that only depends on the *good cache ways*. For the best interval size $T = 160KB$ the LLC performs, on average, twice as good

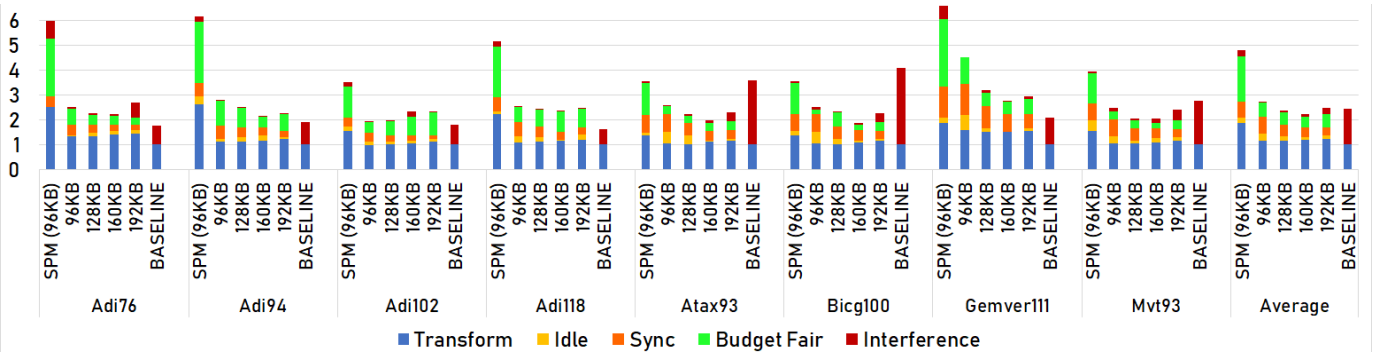

Fig. 6: The results for the individual kernels in *fair* co-scheduling with the CPU.

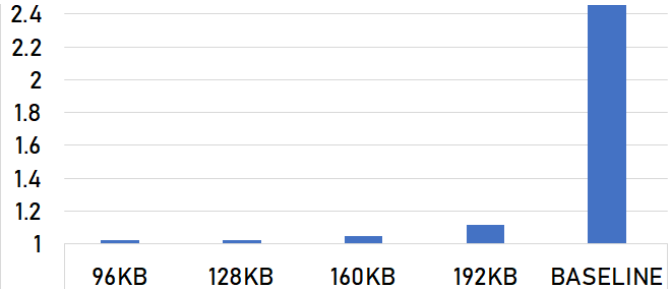

Fig. 7: The average sensitivity to interference for all kernels.

as the SPM, due to the coarser granularity of synchronization made possible by the large LLC. An additional effect of the cache-based approach is that the length of the *M-* and *C-phases* become more balanced, leading to smaller amount of idleness budgeted into the system schedule. In the case of SPMs, the *C-phase* was so short that it would starve the CPU from using memory, but on the LLC, the longer latency to the LLC brings up the execution time, allowing the CPU to get a fair share of memory. The increase in *C-phase* latency is compensated by the lower complexity of the LLC *M-phase* (Figure 2).

We also see that on average, the LLC outperforms the baseline under interference. While the average over these kernels is modest, only 10% improvement, the SPM on average showed a almost a 200% *decrease* in performance, even under interference. However, in the best case, PREM on LLC offers a 215% improvement in WCET compared to the baseline. As the interference to the baseline kernels are found by measurement, they are only a lower bound on the possible slowdown. In contrast, PREM is designed with this limit as a design goal.

B. Predictability

In addition to better performance, the predictability guarantees can still be preserved with the LLC. Figure 7 shows how much, on average, the execution time increases under interference. For intervals similar in size to the SPM, $T \leq 128KB$, the interference only adds 3% to the execution time. For the $160KB$ interval the sensitivity increases to 5% over the execution time in isolation, and for $192KB$, at the limit of the size of the *good ways*, the sensitivity increases further to 15%. However, this is significantly less than the 245% for the unmodified baseline.

VI. Conclusion

State-of-the-art Predictable Execution Models (PREM) are using the scratchpad memory for local storage. However, due to the small size of the SPM it is difficult to hide the synchronization cost inherent to PREM. The solution is to use the larger hardware managed caches, but these have previously been considered too unpredictable for use in timing-critical systems. In this paper, we provide insights on how the caches can be tamed for use with PREM, and show that this provides on average a $2\times$ improvement in performance compared to the SPM-based state of the art, without sacrificing the key goal of predictability.

VII. Acknowledgment

Supported by the EU H2020 project HERCULES (688860).

References

[1] NVIDIA. (2018, Sept) Jetson embedded systems. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

[2] AMD. (2018, Sept) Embedded g-series lx. [Online]. Available: https://www.amd.com/en/products/embedded-g-series-lx

[3] R. Cavicchioli *et al.*, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *ETFA'17*, Sept 2017, pp. 1–10.

[4] H. Yun *et al.*, "Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Trans. on Computers*, vol. 66, no. 7, 2017.

[5] W. Ali *et al.*, "Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms," *ArXiv e-prints*, Dec. 2017.

[6] H. Yun *et al.*, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Techn. and Appl. Symp. (RTAS)*. IEEE, 2013.

[7] R. Pellizzoni *et al.*, "A predictable execution model for cots-based embedded systems," in *RTAS'11*. IEEE, 2011.

[8] A. Alhammad *et al.*, "Time-predictable execution of multithreaded applications on multicore systems," in *DATE'14*. IEEE, 2014.

[9] N. Capodieci *et al.*, "Sigamma: Server based integrated gpu arbitration mechanism for memory accesses," in *RTNS'17*. ACM, 2017, pp. 48–57.

[10] B. Forsberg *et al.*, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *DATE'17*, 2017.

[11] B. Forsberg *et al.*, "Heprem: Enabling predictable gpu execution on heterogeneous soc," in *DATE'18*, March 2018, pp. 539–544.

[12] M. Lv *et al.*, "A survey on static cache analysis for real-time systems," 2016.

[13] X. Mei *et al.*, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[14] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.