An Adaptive Controller to Save Dynamic Energy in LP-NUCA

Darío Suárez Gracia[1], Teresa Monreal Arnal[2], and Víctor Viñals Yúfera[1]

Abstract— **Portable devices often demand powerful processors to run computing intensive applications, such as video playing or gaming, and ultra low energy consumption to extend device uptime. Such conflicting requirements are hard to fulfil and appeal for adaptive hardware that only consumes energy when required.**

LP-NUCA is a tiled cache organization aimed at high-performance low-power embedded processors that sequentially looks up for blocks ordered by temporal locality in groups of small tiles. Unfortunately, LP-NUCA has two main dynamic energy wasting sources: (a) blocks are continuously migrating among tiles even in low locality phases, (b) to reduce cache latency, the tag and data arrays of the tiles are always accessed in parallel.

This paper proposes a learning-based controller that dynamically tunes block migration and cache access policy between parallel and serial. During low temporal locality phases the controller drops blocks from the LP-NUCA root tile, L1, and forces a serial access to the tag and data arrays in the tiles, thus reducing the energy waste. Using a cycle-accurate simulator and energy estimations derived from an LP-NUCA layout, the proposed controller reduces dynamic energy by 20% on average for single and multi-thread workloads.

Keywords— **Cache Hierarchy, Multithreading, Energy, Power, Embedded, NUCA**

I. Introduction

The way people use computers is partially shifting from personal computers with local data to mobile devices with data on the cloud. This “platform” displacement has not carried along “application” changes. Users almost demand the same performance in mobile devices that they used to experiment in desktop computers. Giving the same performance level with the tight energy constraints of mobile environments appeals for adaptive hardware that judiciously detects whether it is profitable to invest energy in order to satisfy the user.

One of the most energy-efficient mechanisms to achieve high-performance is the memory hierarchy [1], where several small caches pretend to be an unbound and fast storage thanks to the locality of programs. Non-Uniform Cache Architecture, NUCA, exploits locality at a finer granularity than conventional caches because they enable inter bank block migrations [2]. Light Power NUCA, LP-NUCA, is a variant of Light NUCA (L-NUCA) for high-performance low-power embedded processors, such as those of mobile devices, that conveys blocks through three specialized Networks-in-Cache as L-NUCA does [3], [4], but also includes two static techniques for saving dynamic energy, Miss Wave Stopping and Sectoring. These techniques together with LP-NUCA ad-hoc network mechanism enable to outperform conventional and static NUCA organizations in terms of energy and performance.

The organization of LP-NUCA consists of many small tiles behaving as a very large distributed victim cache [5]. Blocks remain ordered by temporal locality (TL), so the L1, renamed root-tile (r-tile), recently evicted blocks have a lower service latency than those previously evicted. The LP-NUCA design relies on the temporal locality of programs along all their execution; hence, when the r-tile evicts a block, it triggers a chain of dominoes replacement for maintaining the TL block ordering. But an energy wasting problem can arise during low TL phases. During them, the r-tile floods the rest of tiles with blocks that will be seldom requested. Besides, these blocks degrade older ones that may be re-referenced in the near future. Moreover, LP-NUCA always accesses in parallel tag and data arrays to reduce cache latency. Since a data array access roughly consumes more than 5× the energy of the tag in LP-NUCA [4], this parallel access is a major waste of energy for requests with high likelihood of being a miss. Ideally, we would like to detect low locality phases to prevent the r-tile for evicting low locality blocks and to dynamically switch between parallel and serial access in the rest of tiles.

LP-NUCAs were conceived for single-thread processors; however, to increase their performance⁄energy ratio, current advanced embedded processors rely on extracting parallelism from multiple threads rather than from a single one. For example, the Intel Xeon LC3528, the MIPS MIPS32-1004K, or the Netlogic XLP832 simultaneously execute between 2 and 4 threads [6], [7], [8]. Traditionally, multi-threaded processors (MT) have shared all the cache hierarchy [9] increasing the chances of polluting the cache with useless blocks and evicting useful blocks from other threads. LP-NUCA in MT environments would suffer from this problem and would benefit from a controller able to drop low locality blocks and to retain high locality ones. Finally, in this case we can expect little performance improvements because high locality threads will experiment more hits in the LP-NUCA.

This paper extends LP-NUCA in several significant ways. First, we identify that LP-NUCA wastes dynamic energy during low locality phases by continuously degrading blocks among tiles and by accessing

[1]Computer Architecture Group (gaZ). Dpto. de Informática e Ingeniería de Sistemas. Instituto de Investigación en Ingeniería de Aragón. Universidad de Zaragoza. e-mail {dario, victor}@unizar.es

[2]Department of Computer Architecture. Universitat Politécnica de Catalunya (UPC). e-mail: teresa@ac.upc.edu

the tag and data arrays in parallel even when the likelihood of miss is high. Second, we propose a learning based mechanism based on local search methods that dynamically selects when dropping blocks from the cache will harm neither performance nor energy. Third, we employ the same controller to dynamically adjust between parallel and serial access to the cache tag and data arrays leveraging from the congestion management support from L-NUCA. Fourth, we show that the proposed controller requires minimal hardware for improving energy consumption with small gains in performance.

The rest of the paper is organized as follows. Section II presents the adaptive controller. Section III describes our methodology and simulation environment. Section IV evaluates the results. Section V comments on the related work, and Section VI concludes the paper.

II. Adaptive Drop Ratio Controller

Figure 1 shows the LP-NUCA cache organization. Misses in LP-NUCA operate as follows: when the r-tile misses, it allocates an empty way for the incoming block. When necessary, it evicts a victim block to a neighbour tile with the minimum latency difference. The destination tile, with a transport latency of 3, will repeat the operation to a tile with transport latency of 4, and this dominoes operation continues until a tile has an empty way or a block is evicted from the whole LP-NUCA.

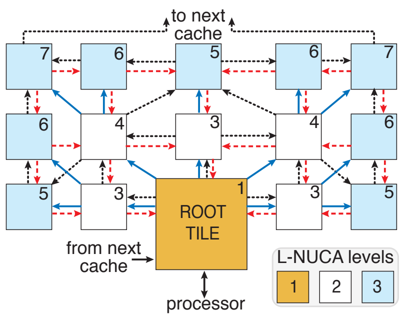


Fig. 1. LP-NUCA basic organization with its three Networks-in-Cache: Search in blue, Transport in red, and Replacement in black. The number in the right upper corner of each tile represents its service latency assuming single-cycle tiles

This chain of evictions keeps blocks ordered by temporal locality but wastes a lot of energy when blocks are not requested before leaving the LP-NUCA. Besides, in their way out, these blocks pollute the tiles and force the evictions of other blocks that may be requested in the near future causing additional damage. The goal of this work is to find a controller minimizing the insertion of these polluted blocks from the r-tile into the rest of tiles. To do so, we propose an Adaptive Drop Ratio controller able to dynamically detect low locality phases of programs and choose the optimal drop rates for all threads in execution. Besides, when dropping all r-tile eviction blocks, the likelihood of a request becoming a cache miss increases, so the controller will switch the access policy of the data and tag arrays inside tiles from parallel to serial to save extra power.

Now, we explain the controller operation. It keeps a reference state with the drop rates of all the threads and its value in the desired target function (cache hits, IPC, . . .). At regular periods, named epochs, the controller changes the drop rate of a single thread (trial), and after N epochs have completed, where N is the number of threads, it ranks the drop rate trials accordingly to the target function. The best trial supersedes the reference when its target value is better than the reference one. To simplify the implementation, the drop rate changes at regular steps, named Δ, and ranges between 0 and 1. A drop rate of 0 means all blocks are evicted to the rest of tiles and of 1 means all blocks are dropped [1]. From a given drop rate, we can move either upwards, adding Δ, or downwards, subtracting Δ. To avoid the trial of both, we add a variable specifying the direction, and restrict the trial to this direction. This variable takes two values, -1 for downwards ($\downarrow$) and 1 for upwards ($\uparrow$), making straightforward the implementation of the controller. In round-robin fashion, the controller selects one thread and computes its trial drop rate as $dr_i^{trial} = dr_i^{ref} + dir_i\Delta$ where dr_i^{trial}, dr_i^{ref}, and dir_i represent the trial drop rate, the reference drop rate, and the direction of thread i, respectively. At the end of thread i trial epoch, dir_i reverts if the reached target is lower than the reference one.

Algorithm 1: Hill-Climbing algorithm of the ADR controller

```
computeEpochStatistics();
if $n\_epoch$ % $n\_threads$ == $0$ then
    foreach thread do
        if not isExempted(thread) then
            if ifTrialBetterThanRef(thread) then
                ref[thread].dir = trial[thread].dir;
            else
                ref[thread].dir = !trial[thread].dir;
            end
        end
    end
    if bestTrialBetterThanRef() or
    maxEpochsWoutChange() then
        refSt.dr = bestTrialSt.dr;
    end
end
n_epoch++;
setTrialState();
```

Algorithm 1 shows the proposed implementation of the ADR controller based on hill climbing with two additional improvements: the exemption of threads and the update of the reference state to avoid temporary maximums. The penalty of dropping useful blocks can be very high because they can cause processor stalls. So when a thread experiences a few number of

[1]Dirty blocks require to be sent to the next cache level in copy-back configurations.

misses, it is counterproductive his evaluation because we will be moving the controller in a flat zone and not towards the steep areas [2]. Regarding the latter, temporary maximums, we observe from the experiments that it is worthy its update either when a trial performs better or after a given number of epochs without change. In this work, we empirically fixed this value to 4× the number of threads in execution.

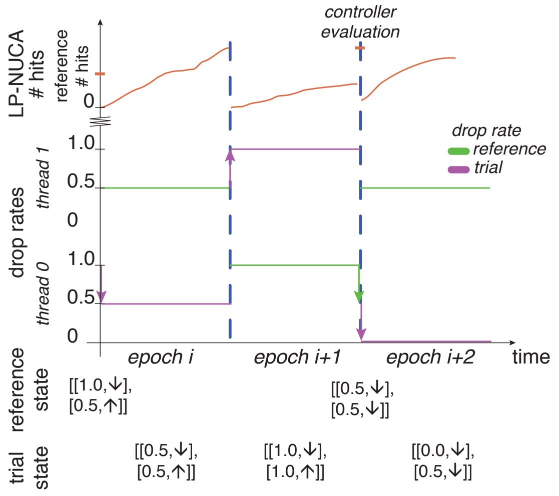


Fig. 2. Temporal behaviour of the Adaptive Drop Ratio Controller with an step, Δ, of 0.5

For example, Figure 2 shows a controller for a 2 thread machine in which the number of LP-NUCA hits is the target function. We assume that the controller state is defined as a list of tuples $ST = [[dr_0, dir_0], ..., [dr_{n-1}, dir_{n-1}]]$. During *epoch i*, *thread 0* is the trial thread and is in trial downwards direction state. Alike, *thread 1* is evaluated in *epoch i+1* with trial upward state. At the end of this epoch, the controller observes the target function, LP-NUCA number of hits, and *epoch i* results better than the reference one. So in *epoch i+2*, dr_1 remains equal and dr_0 reduces in one step keeping the direction. *Thread 1* reverses its direction because in *epoch i+1* the number of hits was lower than in the reference epoch.

TABLE I
Possible organizations for the Adaptive Drop Ratio Controller

Evaluation Trigger	Time	1-512K cycles
	# misses	¼-5× r-tile blocks
Step Size	From 0.1 to 1	
Target Metric	IPC, # hits, reuse rate	
Auxiliary Tags, m	From 0 to 128 entries	
Exemption Threshold	From 5 to 100 MPKI	

To suggest a good ADR controller design based on hill-climbing, we evaluated the various parameters summarized in Table I.

The first big choice is how to trigger a new epoch, either at fixed intervals of time or after a fixed number of r-tile misses. On the former, we have experimented with epochs from 1K to 512K cycles and, on the later from ¼ to 5× the number of blocks the r-tile stores[3]. The second big choice is step size; i.e., with an step of 0.5 a thread can drop all, half, or none of the evicted blocks. Finally, when a thread reaches the "all drop" state, $dr_j = 1$, the controller requires an heuristic for returning the injection of evicted blocks into the rest of LP-NUCA tiles, $dr_j = 0$. One option is to force the return to a state that does not drop all the blocks after a given number of epochs. Another smarter option is the introduction of a small auxiliary tags tracking the last m dropped blocks and lookup for misses in this structure. When updating the controller state if several requests have matched in the auxiliary tags, automatically that thread leaves the "all drop" state. Also, we can fix the misses per epoch that we require to evaluate a thread.

Finally, Figure 3 shows the behaviour of the controller executing 255.vortex with 179.art during 2 millions of cycles. ADR synchronously reevaluates after 4096 cycles, has 3 dropping states, and when the drop ratio is 1, cache arrays are serially accessed. It includes an auxiliary tag array of 512 entries. This configuration was the best among all the test of this work. The plot includes from top to bottom the number of committed instructions, the drop ratio indexes, the number of rest of tiles hits, the number of evicted (inserted) blocks into the rest of tiles, and the number of dropped blocks. 255.vortex, red lines, is an example of a benchmark that is better to exempt from the controller. Its miss rate is very low, and by dropping blocks we could only reduce its performance and increase the accesses to the next cache level. On the contrary, 179.art experiences program phases in which it pollutes the cache. For example, before the 54M point, the controller drop blocks and keeps useful blocks inside the cache that are serviced to the r-tile, and then when the miss rate drops again below the exemption threshold, it evicts all the blocks inside the rest of tiles.

A. Hardware Cost

The hardware implementation of the Adaptive Drop Ratio controller requires minimal overhead. Most current processors already include the performance counters for the target function and we only require to store the reference state for all the threads, 1 bit for the direction plus $\log_2(drop\ ratios)$, and the trial configuration. The partial tags only require an small SRAM array, consuming little energy, and if necessary it could be easily replaced by a bloom filter.

Other novelty of this work is the proposal of switching between parallel and serial access to the cache arrays. At first glance, this feature could be hard to implement, just the opposite is true. The key observation is that when the likelihood of cache miss is high only the tag array is accessed. So we can add an extra bit in the Search Network disabling the accesses to the data arrays in the tiles. Misses will propagate

[2]We could also reevaluate after a number of epochs equals to the no exempted threads, but the hardware complexity will be higher.

[3]Assuming a 32KB r-tile organized in blocks of 32B, there are 1024 blocks in total.

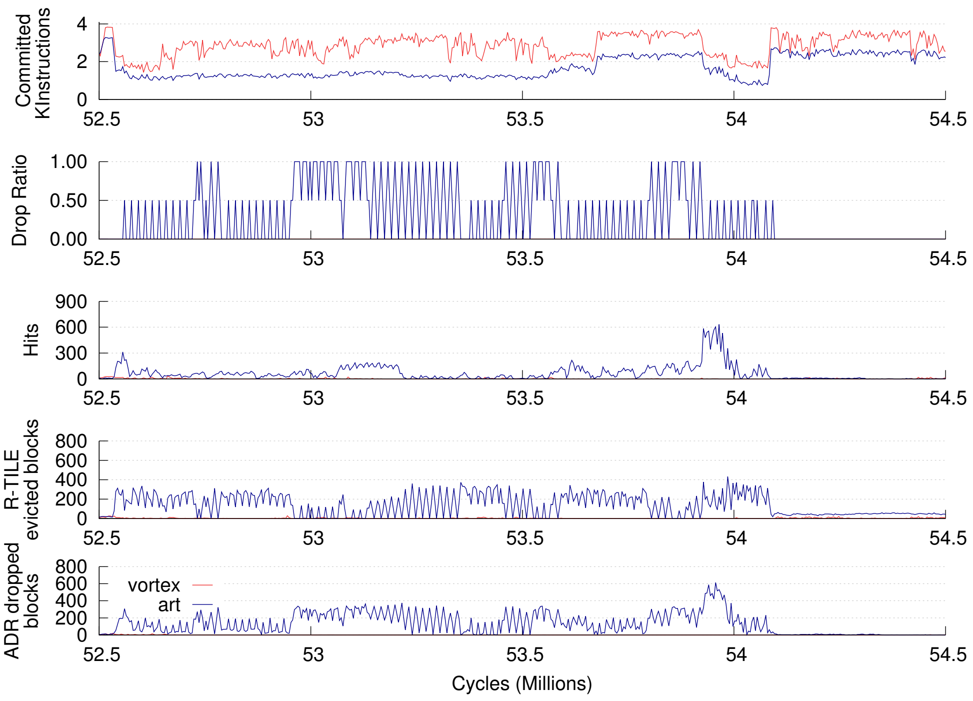


Fig. 3. SMT execution of 255.vortex and 179.art with an ADR of 4Kcycles epochs and 3 drop ratios

back-to-back over the fabric. Nevertheless, in the rare case that a tile hits during a serial access, the data array have to be accessed. Since we can not stop the request propagation in the Search network because it does not have any control flow mechanism, we need to re-inject the request in parallel mode. This re-injection feature is already supported by LP-NUCA to cope with congestion of the Transport network, when a tile hits and does not have any output transport link available. Therefore, a serial request hitting in a tile will reset the serial and set the congestion bits, so that the request be reinserted.

III. Methodology and Simulation Environment

We employ the same simulation environment, energy estimations, and cache hierarchy organizations that previous LP-NUCA work [4]. The baseline processor resembles the IBM/LSI PowerPC 476FP [10], [11] and executes 1 or 2 threads simultaneously. Table II summarizes the main parameters for the reference processor and cache hierarchy, including the same L1 and L3 caches and either a conventional L2, a S-NUCA, or an L-NUCA. All caches use LRU replacement except L-NUCA that employs LRF, least-recently-filled, and they all have a single read/write port.

Our workload comprises the same embedded domain oriented application than previous work [4]. Nevertheless, to get deeper insights from the results, we divide the benchmarks in two groups: low MPKI and high MPKI as table III shows. For the two SMT experiments we present the results in three groups: low MPKI, medium MPKI, and high MPKI when both, one, and none benchmarks of the combination exhibit a low misses per kilo instruction rate.

TABLE II

Simulator Micro-architectural parameters. BS, AM, lat, and init stand for block size, access mode, latency, and initiation rate, respectively

Clock Frequency	1 GHz	Fetch/Decode/ Commit width	2
Issue width	2(IN+ME) +2FP	ROB / LSQ entries	32 / 16
INT/FP/MEM IW entries	8 / 8 / 8	branch predictor	bimodal + gshare, 16 bit
Miss. branch penalty	6	Instruction Cache	perfect
L1/L2/L3 MSHR entries	8 / 8 / 4	TLB miss latency	30
MSHR secon. misses	4	Store Buffer/ L2/ L3 WB size[a]	8 / 4 / 4

L1/r-tile[b]	32KB–4Way–32B BS, write-through, 2-cycle lat, 1-cycle init
L2	512KB–8Way–32B BS, serial AM, 4-cycle lat, 2-cycle init, copy-back
S-NUCA	2×2 128KB–2Way–32BS, parallel AM, 3-cycle lat, 3-cycle init, copy-back
L-NUCA rest of tiles	32KB–2Way–32B BS, parallel AM, copy-back, levels: 3, total size: 448KB

L3	4MB eDRAM–16Way–128B BS, 14-cycle lat, 7-cycle init, copy-back
Main Memory	100 cycles/4 cycle inter chunk, 16 Byte bus

[a] L2, S-NUCA, L-NUCA, and L3 Write Buffers coalesce entries

[b] In r-tile, copy-back and write-around

We used a similar energy estimations than the previous work for the battery powered domain in 32 nm [4]. In the single thread execution, the simulator warms-up caches and branch-predictor for 200M instructions before starting the cycle-accurate simulation. We follow the same approach that Li *et al.* for energy and delay measurements in SMT environments [12], and account for all the energy consumed

TABLE III

Workload selection. *I*, *F*, *0*, *6* refer to Integer, Floating Point, SPEC CPU2000, and SPEC CPU2006, respectively

Low MPKI[a]	186.crafty *I0*, 255.vortex *I0*, 177.mesa *F0*, 458.sjeng *I6*, 482.sphinx3 *F6*
High MPKI	164.gzip *I0*, 179.art *F0*, 187.facerec *F0*, 401.bzip2 *I6*, 445.gobmk *I6*, 464.h264ref *I6*, 473.astar *I6*, 453.povray *F6*

[a] L1 MPKI rate lower than 20 in the baseline processor

until the last thread commits 100M instructions.

IV. Results Evaluation

This section compares the cache organizations presented in previous section, namely, conventional L2 (L2), Static NUCA (SN), LP-NUCA (LP), and two LP-NUCAs enhanced with two adaptive drop ratio controllers: one with synchronous epochs (ADR_C) and another with epoch based on the number of request for evictions (ADR_R). Both ADR_C and ADR_R have been selected after exhaustively exploring the controlling design space with the options shown in Table I. ADR_C re-evaluates the drop ratios and directions every 4096 cycles, while in ADR_R occurs after 1024 attempts of r-tile evictions (or r-tile primary misses). Both controllers share the rest of parameters, namely, 3 dropping states, 512-entry auxiliary tags, and an exemption threshold of 50 MPKI in the rest of tiles.

First, we compare the energy consumption, then we continue with the execution time, and we finish with energy-delay results. For the sake of brevity, we only show overall and averaged results, but the individual behaviours do not differ from the presented one.

A. Energy

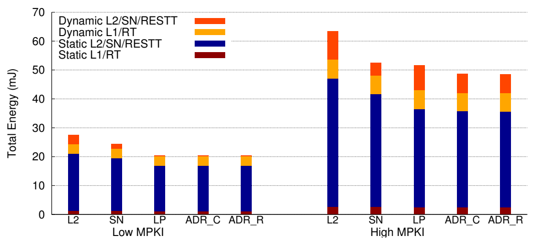
Figure 4 shows the total energy consumed by all configurations. LP-NUCA with the ADR have the best results regardless the benchmark group and the number of threads.

If we focus on the last three bars to compare the performance of the ADR controller we observe that the synchronous one performs slightly better than the based on the number of replacements. In the more energy demanding benchmarks, high MPKI, the controller reduces energy by 20% on average.

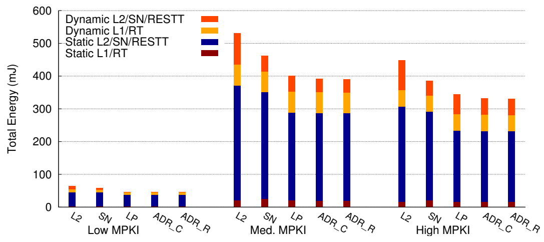
B. Execution Time

Dropping blocks may reduce LP-NUCA hit rates and increase execution time, EX. To verify that the proposed ADRs do not affect EX Figure 5 shows the total execution time.

LP-NUCA shows performance improvements over L2 and SN, and both controllers slightly reduce execution time because they keep inside the cache useful blocks that otherwise would be expelled. ADR_C and ADR_R reduces total execution time by 2.14% and 2.29%, respectively, in the single-thread environment. Improvements become marginal in the 2 SMT and

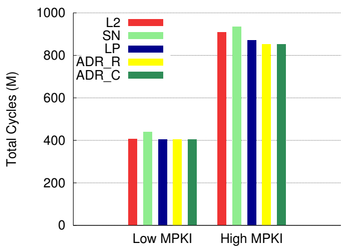


(a) Single Thread

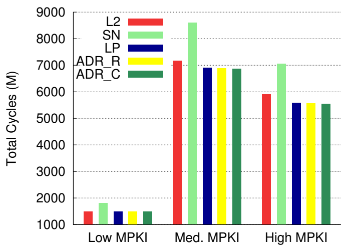


(b) 2 SMT

Fig. 4. Energy consumption comparison



(a) Single Thread



(b) 2 SMT

Fig. 5. Total Execution Time for the different configurations

reduce to 0.62% and 0.89% for ADR_R and ADR_C, respectively, because the multi-threading execution covers the memory stalls.

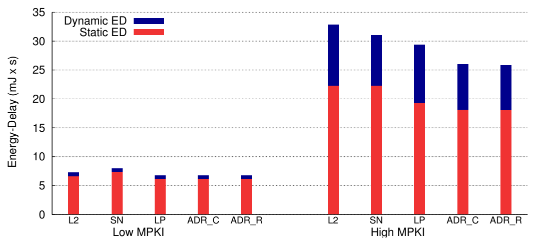
C. Overall System Impact

Finally, we present the sum of the energy-delay of the tested benchmarks. Figure 6 includes the L3 energy to show that dropped blocks are not requested to the L3 increasing the overall energy.

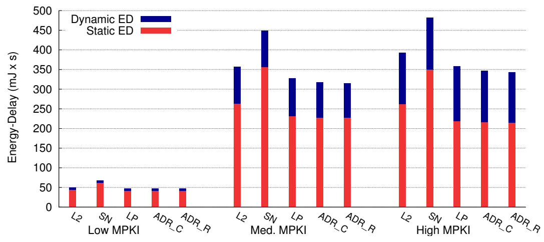
Again ADR_R and ADR_C are the winners in all categories. LP-NUCA improvements in execution time reduces the static component with regards to L2 and SN, and the controllers reduce on average dynamic energy-delay, their target, 7.6% for all but low MPKI workloads.

V. Related Work

Architects have proposed a plethora of designs to save cache energy through reconfigurable caches that



(a) Single Thread



(b) 2 SMT

Fig. 6. Energy-Delay. This figure includes L3 cache consumption as well

change their number of ways, sets, or both at run time [13], [14], [15], [16]. For an updated state of the art please refer to Sundararajan *et al.* [16]. Previous works adapt the cache at a finer granularity than this work, and most proposed techniques can be easily applied to the LP-NUCA. Contrary to the original LP-NUCA design [4], this work proposes a proactive dynamic technique to save energy while previous ones, Sectoring and Miss Wave Stopping, were completely static and application agnostic. Besides, this work analyzes SMT workloads which have not been extensively studied.

Regarding the learning based approach, the Hill Climbing algorithm has been employed for distributing resources in SMT processors [17], but not for cache reconfiguration.

VI. Conclusions

Ultra-portable mobile devices demand quasi-desktop performance with a fraction of energy consumption. Since application behaviour changes during execution, processors require adaptive mechanism wasting the minimum amount of energy when necessary.

This paper proposes an adaptive controller for LP-NUCA, a tiled organization for high-performance low-power processors, that automatically decides when cache blocks are not reused and can be dropped reducing the cache activity. Besides, during high dropping phases, the controller is able to change the cache array access from parallel to serial further reducing the energy consumption.

With representative workloads, a cycle-accurate simulator, and implementation based energy estimations, we observe that the proposed controller reduces dynamic energy on average by 20% for single-thread and 2-threaded workloads without increasing the execution time.

Acknowledgement

The authors would like to thank Luis Montesano del Campo for his helpful comments on the learning strategies. This work was supported in part by grants TIN2010-21291-C02-01 and TIN2007-60625 (Spanish Government), gaZ: T48 research group (Aragon Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

References

[1] Shekhar Borkar and Andrew A. Chien, “The future of microprocessors,” *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[2] Changkyu Kim, Doug Burger, and Stephen W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *Proc. of ASPLOS-X*, 2002.

[3] Darío Suárez, Teresa Monreal, Fernando Vallejo, Ramón Beivide, and Víctor Viñals, “Light NUCA: a proposal for bridging the inter-cache latency gap,” in *Proc. of DATE'09*, 2009.

[4] Darío Suárez Gracia, Giorgos Dimitrakopoulos, Teresa Monreal Arnal, Manolis G.H. Katevenis, and Víctor Viñals Yúfera, “LP-NUCA: Networks-in-Cache for high-performance low-power embedded processors,” *to appear in IEEE Trans.on VLSI Systems*, 2011.

[5] Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proc. of ISCA'90*, 1990.

[6] Intel Embedded, “Intel® Xeon® processor C5500/C3500 series. Datasheet–Volume 1,” February 2010.

[7] MIPS Technologies, “MIPS32® 1004K™coherent processing system (CPS),” 2010.

[8] Tom R. Halfhill, “Netlogic broadens XLP family,” *Microprocessor Report*, vol. 24, no. 7, pp. 1–11, 2010.

[9] D.M. Tullsen, S.J. Eggers, and H.M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *Proc. of ISCA'95*, 1995.

[10] Tom R. Halfhill, “The rise of licensable SMP,” *Microprocessor Report*, vol. 24, no. 2, pp. 11–18, 2010.

[11] LSI Corporation, “PowerPC™ processor (476FP) embedded core product brief, `http://www.lsi.com/DistributionSystem/AssetDocument/PPC476FP-PB-v7.pdf`,” January 2010.

[12] Yingmin Li, David Brooks, Zhigang Hu, Kevin Skadron, and Pradip Bose, “Understanding the energy efficiency of simultaneous multithreading,” in *Proc. ISLPED'04*, 2004.

[13] David H. Albonesi, “Selective cache ways: on-demand cache resource allocation,” in *Proc. of MICRO'32*, 1999.

[14] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas, “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures,” in *Proc. of MICRO'33*, 2000.

[15] Chuanjun Zhang, Frank Vahid, and Walid Najjar, “A highly configurable cache architecture for embedded systems,” in *Proc. of ISCA'03*, 2003.

[16] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham, “Smart cache: A self adaptive cache architecture for energy efficiency,” in *Proc. of SAMOS'11*, 2011.

[17] Seungryul Choi and Donald Yeung, “Hill-climbing smt processor resource distribution,” *ACM Trans. Comput. Syst.*, vol. 27, no. 1, pp. 1–47, 2009.