REDUCING CACHE HIERARCHY ENERGY CONSUMPTION BY PREDICTING FORWARDING AND DISABLING ASSOCIATIVE SETS

PABLO CARAZO

RUBÉN APOLLONI

FERNANDO CASTRO , DANIEL CHAVER , LUIS PINUEL
and FRANCISCO TIRADO

The first level data cache in modern processors has become a major consumer of energy due to its increasing size and high frequency access rate. In order to reduce this high energy consumption, we propose in this paper a straightforward filtering technique based on a highly accurate forwarding predictor. Specifically, a simple structure predicts whether a load instruction will obtain its corresponding data via forwarding from the load-store structure — thus avoiding the data cache access — or if it will be provided by the data cache. This mechanism manages to reduce the data cache energy consumption by an average of 21.5% with a negligible performance penalty of less than 0.1%. Furthermore, in this paper we focus on the cache static energy consumption too by disabling a portion of sets of the L2 associative cache. Overall, when merging both proposals, the combined L1 and L2 total energy consumption is reduced by an average of 29.2% with a performance penalty of just 0.25%.

1. Introduction

Continuous technical improvements in the current microprocessors field lead the trend toward more sophisticated chips. Nevertheless, this fact comes at the expense of significant increase in energy consumption, which jeopardizes the architects' goal of simultaneously delivering both high performance and low energy consumption. In order to mitigate this problem, many researchers have focused their efforts on reducing the overall energy dissipation in an out-of-order processor. It can be argued that this research problem is not a major concern now due to the trend towards multi-core architectures made by the industry, in which in some cases the pipelines employed are simpler. However homogeneous multi-manycore architectures with in-order pipelines will only provide substantial benefits for scalable applications/ workloads, and some researchers have recently highlighted that future designs will benefit from asymmetric architectures that combine simple and energy-efficient cores with a few complex and energy-hungry cores.[1] The local inefficiencies of a complex core can translate into global performance/per-watt improvements since a complex core could accelerate the serial phases of applications when the energy-efficient cores are idle. This way, a single chip will be able to provide good scalability for parallel applications as well as ensure high serial performance. In summary, as promoted in Ref. 2, researchers should still investigate methods of improving sequential performance despite we have entered into the multicore era.

In an out-of-order microprocessor energy dissipation is spread across different structures including caches, register files, the branch predictor, etc. Specifically, on-chip caches consume a significant part of the overall energy by themselves (see Refs. 3–9). This energy consumption is divided between active or dynamic energy, which is the energy used while the product is performing its various functions, and leakage or static energy, which is the energy consumed by unintended leakage that does not contribute to the integrated circuit (IC) function. Furthermore, it is worth noting that leakage energy has become a top concern for IC designers in deep sub-micron process technology nodes (65 nm and below) because it has increased to a significant percent of the total IC energy consumption. In this paper we intend to reduce the cache hierarchy energy consumption in an out of order processor by decreasing the L1 data cache (DL1) dynamic contribution as well as the L2 cache static part.

The first mechanism that we propose — oriented to reduce the DL1 dynamic energy consumption — is based on an efficient management of the LSQ (load-store queue) and DL1 accesses. One of the main LSQ tasks is to supply the correct data to load instructions via a forwarding process — store to load forwarding — ruling out the cache data and therefore turning the cache access unnecessary. Taking advantage of the cached load-store queue (CLSQ) proposed by Nicolaescu *et al.*[10] — where the number of loads that receive their data from a previous store augment considerably — and using an accurate forwarding predictor that suggests if a load

instruction is likely to receive its data through forwarding, we manage to filter many accesses to the data cache in the target platform — an x86 architecture — while the performance delivered remains largely unchanged.

Our second proposal is focused on reducing the static energy consumption in the cache hierarchy. To this end we disable some of the sets in the second level cache according to the cache array geometry and we analyze the consequent impact on performance and global energy consumption.

The rest of the paper is organized as follows. Section 2 recaps related work. Sections 3 and 4 bring in our two proposals. Section 5 details our experimental environment, while Sec. 6 outlines experimental results and analyses. Finally, Sec. 7 concludes.

2. Background

Many techniques for reducing the cache energy consumption have been explored recently. Next, we recap some of the more outstanding ones. One alternative is to partition caches into several smaller caches[11] with the corresponding reduction in both access time and energy cost per access. Another design, known as filter cache,[12] trades performance for energy consumption by filtering cache references through an unusually small L1 cache. An L2 cache, which is similar in size and structure to a typical L1 cache, is placed after the filter cache to minimize the performance loss. A different alternative, named selective cache ways,[13] provides the ability to disable a subset of the ways in a set associative cache during periods of modest cache activity, whereas the full cache will be operational for more cache-intensive periods. Another different approach takes advantage of the special behavior in memory references: we can replace the conventional unified data cache with multiple specialized caches. Each one handles different kinds of memory references according to their particular locality characteristics.[14] These alternatives make it possible to improve in terms of performance or energy efficiency. Finally, Jin *et al.*[15] obtain energy savings in L1 cache by exploiting loads spatial locality. In their technique, loads always bring a macro data from the processor cache, allowing additional opportunities for load to load forwarding. Nicolaescu *et al.*[10] propose to avoid the data cache access for those loads that receive their data through forwarding. To increase the amount of this kind of loads, they modify the LSQ design to retain load and store instructions after their commit phase. Thereby, a later load in program order augments the chances of obtaining its data from a previous instruction, either an in-flight store, a committed store, or a committed load (load to load forwarding). The mechanism — named *cached load store queue*, *CLSQ*, made of the *CLQ* and the *CSQ* — is based on the low observed rate of LSQ occupancy for some program phases, which make it possible to earmark unoccupied entries to already committed load or store instructions. Our work serves of CLSQ to augment the amount of forwarding loads and as a result to reduce the cache

hierarchy energy consumption, improving and significantly extending our previously published work.[16]

As in our first proposal we are using a forwarding predictor, we should mention that many proposals relying on memory dependence prediction to know in advance which pairs of store-load instructions become dependent[17,18] exist. However, they all exceed the goal of our job. Finally, there are some techniques oriented to selectively disable a portion of cache to reduce the static energy consumption, like gated-Vdd,[19] and others, like Refs. 20 and 21, that we will describe in detail in Sec. 4 for a better understanding of our second proposal.

3. Reducing Dynamic Energy Consumption in DL1

3.1. *Rationale*

In most conventional microprocessors each load instruction consults the first level data cache in order to move the required data into an available register. Simultaneously, the store-queue (SQ) is searched looking for a previous matching in-flight store. If it is found, the store forwards the corresponding data. Otherwise, the data is provided by the cache (see Fig. 1, Original Architecture).

The first technique that we propose in this paper is based on the observation that if a load obtains the corresponding data directly from an earlier store, then the data cache access turns completely unnecessary, so it could be avoided for saving some energy. Obviously, this energy reduction will become significant only if the amount of loads that get the data from the SQ is high enough.

In a RISC processor, the amount of store-load forwarding is relatively small (less than 15% on average according to Ref. 22), basically due to the fact that the number of architectural registers is commonly set to 32 and a register–register architecture is generally implemented. In such scenario, the benefits of trying to avoid the DL1 access could turn meaningless. However, in a register–memory architecture with only 16 architectural registers — as in the case of x86-64, the architecture employed in this job — the number of store to load forwardings is considerably higher as a result of the extra operations due to register spilling.

In a complementary way, we can use Nicolaescu's CLSQ from Ref. 3, which significantly increases the number of loads that receive their data via forwarding, both due to store to load forwarding from the Cached-SQ and to load forwarding from the Cached-LQ.

In summary, in an x86-64 architecture using Nicolaescu's Cached-LSQ, the amount of forwarding can be relatively high — up to 40% of the loads — which makes our idea about saving energy appealing. However, in order to filter the accesses to DL1 performed by loads that may obtain the data directly from the LSQ, we need to either serialize the LSQ and DL1 searches, or to know in advance — i.e., make a prediction — whether the load will obtain the data via forwarding or not. This is a key issue that we address in the next section.

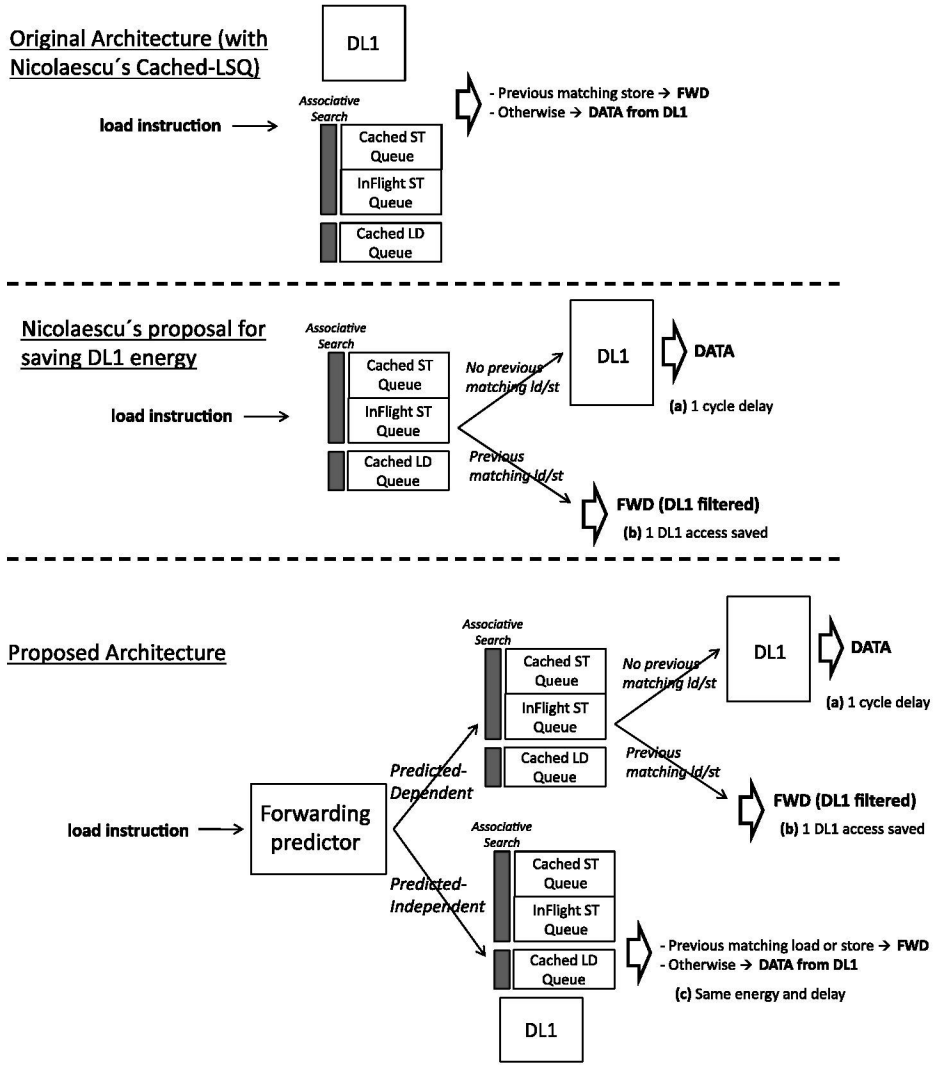

Fig. 1. Original architecture (with the CLSQ), Nicolaescu's architecture and our proposed architecture.

3.2. *Overall structure*

As mentioned above, an obvious implementation would be to serialize the accesses involved (like Nicolaescu does in Ref. 10): the load first scans the SQ, and then — only when the data is not found — a cache search is performed (see Fig. 1, Nicolaescu's Proposal). However, this design is not efficient: when a previous matching store is not found, the delay incurred in accessing to the data cache will result in a significant slowdown. In this paper we will turn up with a much more convenient approach.

Our proposed design (see Fig. 1, Proposed Architecture) is based on a forwarding predictor, which tries to identify those loads that will receive the corresponding data via forwarding. For convenience of discussion, we loosely refer to these loads as *predicted-dependent* loads and the remainder *predicted-independent* loads.

For predicted-dependent loads, only the entire SQ — both conventional and cached parts — and the cached-LQ are searched, omitting the DL1 access (obviously at the risk that the predictor was wrong, in which case the cache access is launched with a one-cycle delay). For the remaining loads the entire SQ, the cached-LQ and the DL1 are searched in parallel (note that in this case, if the predictor fails the performed data cache access turns unnecessary). A predictor with high accuracy provides significant energy savings at the cost of tiny performance degradation.

3.3. *Forwarding predictors*

There is a whole lot of research in the field of memory dependence prediction. However, most proposals employ sophisticated predictor structures, which are excessive for our goal of just predicting in advance whether a load will obtain the corresponding data via forwarding. For this reason, we have not considered them in this paper. Instead, we have evaluated two kinds of simple predictors: Bloom Filter-based[23] and Branch Predictor-based.[24]

Bloom Filter-Based Predictor: This first kind of predictor is a low-overhead table of counters. When a memory instruction — load or store — is issued, it accesses the table based on its address and increments the corresponding counter. Besides, and before incrementing the counter, load instructions check the table to obtain the prediction: if the corresponding counter is greater than zero, then potentially one or more memory instructions access the same memory location, so the systems conservatively predicts the load to obtain the corresponding data via forwarding. Otherwise, the load is predicted-independent.[a]

Branch Predictor-Based: The second kind of predictor is based on the well-known bimodal branch predictor. Similarly to branch instructions, a large majority of loads have a strongly biased behavior — they either frequently or almost never receive the corresponding data via forwarding — so such a predictor performs satisfactorily. A benefit of this Bimodal Predictor compared to the Bloom Filter-based predictor lies on the fact that the prediction is available as soon as the load instruction is decoded. On the contrary, the Bloom Filter is consulted using the load memory address, which needs to be calculated first, so the availability of the prediction is delayed until the issue phase.

[a] As explained in Ref. 25, the SQ and LQ accesses could be avoided in this case. However, since a DL1 cache access is much more energy consuming than an LQ–SQ access, in this job we do not consider such LQ or SQ filtering capability, which would require a deeper study.

Combined Predictor: Finally, we should mention that we have also considered in our evaluation a combined predictor, merging a Bloom Filter with a Bimodal predictor. For extracting the final prediction, a load is marked as predicted-dependent only when both structures predict the load to be dependent. Such a structure benefits from both past forwarding behavior and memory address information, providing the best results as it will be detailed in the evaluation section.

3.4. *Supporting coherence and consistency*

The LSQ from the baseline architecture receives invalidation requests from remote processors, so coherence and consistency functionalities can easily be supported in our technique. However, we should highlight a conflict situation that turns up in our design when implemented in a system with a MESI coherence protocol: if a data is replaced from the DL1 but remains in the Cached-LSQ, the Shared Line will not be activated due to a remote read request, potentially putting the remote data in an erroneous Exclusive State (instead of a Shared State). A possible solution is to force the LSQ to activate the Shared line for every remote read to a load whose data was received via forwarding. As a future work we intend to improve this management since — although straightforward — it is relatively inefficient.

4. Reducing Static Energy Consumption in L2

Static energy is consumed when transistors in the chip remain in the steady state. The relative weight of this kind of energy over the overall consumption has significantly augmented as the technology scales down, at least until the Intel High-K metal gate transistor appearance. In cache structures it is usual to reduce this contribution by *turning off* some associative sets or some cache ways that are considered as dispensable. In this context, we mean by *turning off* to reduce the transistors source-drain voltage drop, thus decreasing the leakage currents involved. Two basic approaches exist for this purpose: Stacking Effect-based and Drowsy Effect-based.

Transistor stacking refers to the technique of stacking off transistors source to drain. Stacked off transistors significantly restrict the leakage current flowing to ground. This is because the voltage differential between the drain and source of the stacked transistors is less than V_{dd}. A popular stacking mechanism is the gated-V_{dd} technique developed by Powell *et al.* for memory cells, which has been successfully employed in many architectural techniques, such as the DRI I-cache[19] and cache decay[20] among others. These techniques are collectively known as nonstate-preserving (or state-destroying): the cell quickly loses its stored value going into a limbo state. Restoring the power supply (turning on the sleep transistor) allows the internal nodes of the cell to recharge, but they take on a random logic state.

In response to the gated-V_{dd} problem of losing state, Flautner *et al.* proposed another approach to curb leakage in memory cells.[21] The drowsy mode is a low

supply voltage mode for the memory cells: memory cells which are idle, i.e., are not actively accessed, can be voltage-scaled into a drowsy mode. In this mode, transistors leak much less than with a full V_{dd}. A "drowsy" bit controls the two levels of supply voltage (V_{dd} or V_{ddLow}) to the memory cells of a cache line. Memory cells are in drowsy mode when fed from V_{ddLow}. The leakage reduction of the drowsy mode is not as profound as that of the gated-V_{dd} approach, but in return the state of the memory cell is preserved. However, a memory cell in drowsy mode cannot be accessed with the full-V_{dd} circuitry of the cache. It first has to be voltage-scaled back to full V_{dd}. Because this is not instantaneous, there is a penalty, albeit small, in accessing drowsy cells.

In this work, based on the observation that the second level cache in a conventional processor is underutilized in some benchmarks (as it will be stated in the evaluation section), we propose to partially turn off this level during the whole application. In order to determine the amount of sets to disable in each application, we look for a satisfactory energy-performance trade-off. Specifically, we select the configuration that exhibits the highest energy savings without degrading performance beyond 1%. Besides, as the disabled zone remains turned off during the entire execution — and hence data preservation is not required — we employ a Stacking Effect-based technique.

Both cache ways and associative sets could be disabled in a selective L2 turning off. Nevertheless, according to CACTI 5.3,[26] the bits from a cache line are spread along a subarray row, as Fig. 2 illustrates.

Thus, only disabling set is a feasible choice. Specifically, for our L2 configuration (256 KB size, 64 B line, 16 ways, 1 bank and 1 r/w port), the parameters associated with the data array geometry are: Ndwl = 32, Ndbl = 4 and Ndsp = 1. With this kind of geometry it is feasible to turn off associative sets by 64 size modules, i.e., we use L2 caches with 256 (baseline), 192, 128 and 64 associative sets.

Finally, it is worth to note that coherence and consistency can easily be supported in this design: at the beginning of an application, we decrease (disabling sets) or

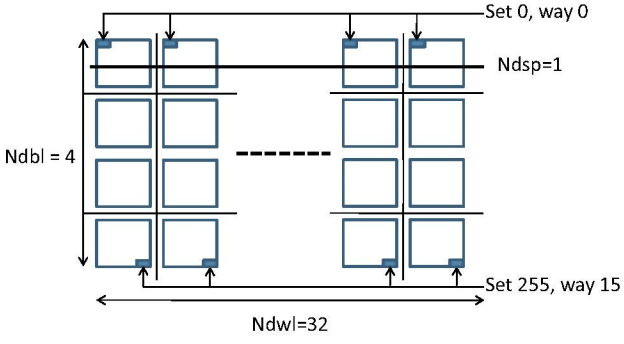

Fig. 2. L2 data array geometry according to CACTI 5.3.

increase (enabling sets) the L2 cache size. In the first case, we flush to the next cache level (L3) data from the disabled sets, whereas in the second case we flush to L3 data from the sets that are active. Given that we only perform these operations once per application, the impact on performance and energy consumption is almost negligible.

5. Experimental Framework

We have evaluated our proposed design using PTLsim,[27] a performance-oriented simulation tool. The simulated microarchitecture follows the default PTLsim configuration that results from the merging of different features of an Intel Pentium 4,[28] an AMD K8 and an Intel Core 2.[29] Some of the main simulation parameters are listed in Table 1.

The evaluation of our proposals is performed using 28 benchmarks from the SPEC CPU2006 suite,[30] compiled for the x86 instruction set. The technology parameters correspond to 45 nm, with a 1.0 V V_{dd}. We simulate regions of 400M instructions after reaching a triggering point, which marks the beginning of code area in which the application behavior is representative of the overall execution.

To evaluate the impact of our data cache filtering and our L2 turning off mechanism over the energy consumption of the cache hierarchy, we use CACTI 5.3 to model the caches of Table 1. Specifically, in order to estimate the caches energy consumption, we have developed an exhaustive energy model that will be detailed next. Furthermore, the simulator has been modified to incorporate in the microarchitectural simulation the predictors from our first proposal, although their energy consumption is considered as negligible compared with the energy savings obtained in the DL1.

In the next three subsections we describe how we determine the triggering point to be employed for each benchmark (Sec. 5.1), we analyze the reliability of the data our simulations report (Sec. 5.2) and finally we detail the energy model used (Sec. 5.3).

Table 1. Simulation parameters for default PTLsim configuration.

Processor	Caches
— Branch predictor: Combined (Bim-2bits + Gshare), BTAC 2k — Instruction fetch queue size: 32 — ROB size: 128 — LSQ size: 80 (LQ: 48, SQ: 32) — Physical registers: 128 — Functional Units: 8, 4 ALU (2 INT, 2 FP), 2 Load, 2 Store — Fetch/Issue/Commit width 4/4/4	— L1 Instruction cache: 32 KB (4 way, 64 B line), 1 bank, 1 port — L1 Data cache: 16 KB (4 way, 64 B line, 2 cycles latency), 1 bank, 1 port — L2 cache: 256 KB (16 way, 64 B line, 6 cycles latency), 1 bank, 1 port — L3 cache: 4 MB (32 way, 64 B line, 16 cycles latency), 2 banks, 1 port
	Memory
	— Main memory latency: 140 cycles

5.1. *Determining the triggering point for each application*

As mentioned above, we simulate regions of 400M instructions after reaching a triggering point, hoping that the selected simulation window deliver almost identical results than those derived from the whole benchmark execution. We have analyzed exhaustively how to determine this optimal point. To this end, we employ a methodical process based on the *gprof* tool[31] and direct observation, contrasting the obtained results with those provided by the SimPoint tool.[32] The rationale behind this procedure is that as SimPoint provides several triggering point per benchmark and they cannot be extrapolated to be used in PTLsim, we try to determine a single and representative triggering point per application.

As the simulation environment is relatively new, the simulations were made very meticulously. First, using the information provided by *gprof* as well as a manual track with *ad hoc* timing measurements, we managed to identify the dominant phase for each application. Second, we employ simulation windows of 100, 200, 400 and 1,000 million instructions within the identified dominant phase trying to obtain an approximation to the mean value of the required metric.

Finally, we choose a single triggering point per benchmark which delivers a mean value as close as possible to the mean value obtained using a more elaborated process.

Specifically, we are interested in determining the forwarding rate for each application under study. Thus, knowing the number of loads that receive the corresponding data from the LSQ instead of from the data cache is desired in order to calibrate the potential benefits of our first proposal. The triggering points experimentally identified for the SPEC CPU2006 benchmarks are detailed in Table 2.

In order to determine the optimal length for the simulation window, we analyze the forwarding data obtained in runs that, starting from the triggering points detailed in the table above, execute a different number of instructions.

The mentioned data are shown in Table 3, where the percentage of loads that receive the corresponding data via forwarding in each application is illustrated. We provide the forwarding rate observed with and without using the XEN[33] hypervisor — an infrastructure that provides full system x86-64 simulation, not only user space — over the architecture of Table 1, as well as that when an extended architecture (increasing processor resources) is employed. As shown, the forwarding rate remains quite stable across the different simulation window lengths, so based on simulation time requirements and simplicity, we finally choose 400M as the number of instructions to be executed after each triggering point is reached without XEN hypervisor. Besides, three applications (*gobmk*, *sjeng* and *dealII*) were discarded due to simulator failures and hence not considered from now on.

To justify and validate our simulation method we contrasted our forwarding rate results with those obtained when the SimPoint tool[32] is employed. We applied the mentioned tool to most benchmarks (some mistakes appeared for a handful of them), obtaining the starting addresses of the most representative execution phases. It is

Table 2. Triggering points for SPEC CPU2006 applications.

Bench	File	Line	Function	Bench	File	Line	Function
perlben.	perlmain.c	97	perl_run (my_perl)	zeusmp	lorentz.f	421	first instruction
bzip2	spec.c	332	spec_ compress	gromacs	md.c	407	while (...)
gcc	toplev.c	2405	if (...)	cactus	PUGH/ Evolve.c	88	while (...)
mcf	mcf.c	157	global_opt	leslie3d	tml.f	324	DO WHILE (...)
gobmk	interface/ main.c	828	Switch (playmode)	namd	spec_namd.c	181	for (...)
hmmer	hmmsearch.c	621	while (...)	dealII	step-14.cc	4060	for (...)
sjeng	search.c	1656	for (...)	soplex	spxsolve.cc	186	do {...} while (...)
libquan.	shor.c	103	quantum_exp_ mod_n	povray	render.cpp	1327	for (...)
h264ref	lencod.c	307	for (...) *[iter. 2]*	calculix	nonlingeo.c	773	while (...)
omnetpp	libs/cmdenv/ cmdenv.cc	293	if (...)	GemsF.	leapfrog.f90	231	CALL UPML updateE
astar	Library.cpp	293	for (...)	tonto	mol.F90	8726	select case (...)
bwaves	shell_lam.f	297	call bi_cgstab_ block	lbm	main.c	28	for (...)
gamess	rhfufh.F	2223	DO 300 ITER = 1, MAXIT	wrf	solve_em.F90	532	DO
milc	gauge_stuff.c	227	for (...)	sphinx3	spec_main_ live_ pretend.c	166	for (...)

Table 3. Forwarding rate (%) for SPEC CPU2006 applications.

	With XEN				Without XEN		Without XEN[a]
Benchmark	100M	200M	400M	1000M	100M	400M	400M
400.perlbench	34.6	36.3	36.7	43.8	34.4	37.1	45.9
401.bzip2	49.5	45.0	42.3	42.7	49.7	43.3	51.2
403.gcc	44.7	47.0	47.9	46.7	46.2	49.1	59.5
429.mcf	22.8	22.8	24.1	24.7	22.6	25.0	27.8
445.gobmk	44.8	44.9	43.8	43.1	45.1	[b]	[b]
456.hmmer	27.3	27.0	26.9	26.7	35.2	35.4	55.7
458.sjeng	34.1	34.1	34.3	34.1	35.3	35.5	45.8
462.libquantum	0.0	0.0	0.0	0.0	0.0	0.0	0.0
464.h264ref	26.1	27.7	28.6	28.6	26.1	28.4	33.3
471.omnetpp	36.5	36.2	36.2	35.2	36.6	36.2	42.3
473.astar	60.9	61.6	61.6	61.9	63.9	61.3	74.7
410.bwaves	25.7	26.6	26.8	26.1	23.1	24.7	27.6
416.gamess	36.2	34.9	34.3	33.8	37.1	35.2	54.1
433.milc	44.4	49.8	48.0	43.0	47.0	44.2	31.6

Table 3. (*Continued*)

Benchmark	With XEN				Without XEN		Without XEN[a]
	100M	200M	400M	1000M	100M	400M	400M
434.zeusmp	56.1	60.3	61.5	61.0[c]	58.7	61.8	51.4
435.gromacs	41.4	39.8	38.8	37.8	48.7	45.0	59.5
436.cactusADM	25.9	25.7	25.6	25.5	26.3	25.6	38.8
437.leslie3d	28.7	29.1	34.7	32.9	30.4	35.7	38.1
444.namd	33.3	33.5	33.6	33.6	35.0	35.2	44.4
447.dealII	15.0	14.2	[d]	[d]	17.2	[d]	24.0[e]
450.soplex	41.7	41.2	41.2	41.3	47.7	46.7	51.3
453.povray	45.3	45.1	45.2	45.1	46.7	46.7	57.5
454.calculix	45.6	45.8	45.8	45.9	45.0	45.0	49.4
459.GemsFDTD	9.9	9.5	9.4[f]	[f]	10.7	9.5	18.3
465.tonto	37.3	37.2	37.4	37.3	40.8	39.1	52.2
470.lbm	54.8	55.0	55.4	55.3	51.1	52.1	48.7
481.wrf	32.1	38.9	28.8	30.2	33.3	27.9	29.2
482.sphinx3	10.2	10.5	10.7	10.0	10.4	10.4	16.9
AVERAGE	**34.5**	**35.0**	**34.8**	**34.6**	**35.9**	**35.7**	**41.8**

[a]Using an extended architecture: ROB size: 256, LSQ size: 160 (96/64), Issue Queue size: 32, Number of physical registers: 256.
[b]Simulation failure in ooocore.cpp:1095.
[c]With 983M instructions the application is executed until completion.
[d]Simulation problems appear after 300M instructions.
[e]Forwarding rate with a 200M instruction simulation window due to problems appeared after 300M.
[f]With 399M instructions the application is executed until completion.

worth noting that we collect a total amount of 215 triggering points for 24 benchmarks, i.e., 9 points per application on average, although some applications, like *omnetpp* and *cactusADM*, just exhibit 1 and 2 triggering points, respectively, whereas for other benchmarks, like *zeusmp*, 15 points were detected. For each triggering point the forwarding rate was measured, although in an approximated fashion since, due to a simulation framework constraint, a triggering point was established when the address of this point is reached for the first time. In Fig. 3 we show

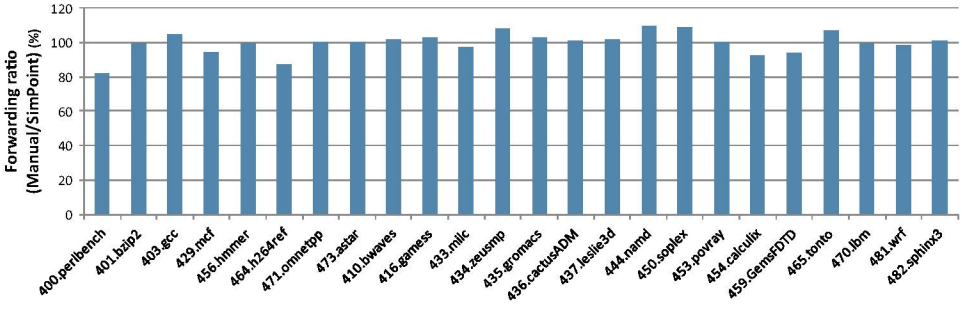

Fig. 3. Forwarding ratio (%) for SPEC CPU2006 applications (Manual value/SimPoint value).

a comparison between the load forwarding rate reported by our method and that when using SimPoint. Specifically, the figure illustrates, for each application, the percentage that the forwarding rate obtained with our simulation method represents with respect to the forwarding rate reported by the SimPoint tool.[32] As illustrated, there is a high correlation degree between data reported by the two methods.

5.2. *Reliability of reported data*

As it will be detailed in the evaluation section, we carried out many simulations changing some microarchitectural details and configuration parameters in order to determine how these changes impact on performance and energy consumption. It is important to discriminate between which part of IPC and energy savings is related with the intrinsic variability associated to different execution runs and which one is derived from changes in the processor microarchitecture.

For this purpose we repeat, over the same machine, a whole simulation of all benchmarks to obtain four measurements per application (using a simulation window of 400M instructions). Figure 4 illustrates the range of fluctuation (percentage) in the number of execution cycles with respect to the arithmetic mean obtained from the four runs. As shown, most applications exhibit a high stability. Only *astar* moves away from this trend, so it is not included in the following experiments.

We also analyze the variability associated with energy model parameters (they will be described in detail in Sec. 5.3). Again we repeat (four times) a whole simulation of all benchmarks over the same machine, using a simulation window of 400M instructions. Figure 5 illustrates the average variability for each parameter. Although few benchmarks exhibit moderate variability for some particular parameters, average values considering all applications, as shown, remain notably low.

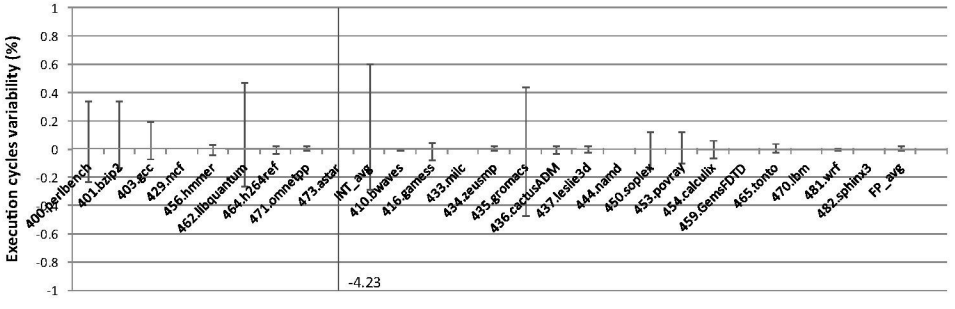

Fig. 4. Execution cycles variability (simulation runs over the same machine).

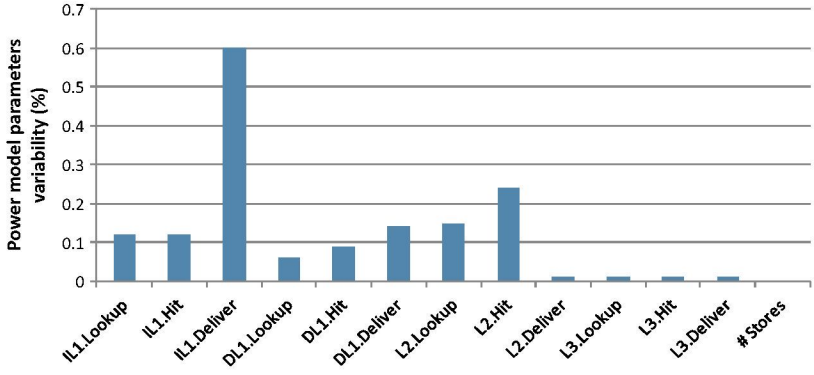

Fig. 5. Variability of energy model parameters.

5.3. *Energy model*

In the evaluation section we report data about the energy impact of our proposals over the cache hierarchy. In order to accurately provide these results, we developed an energy model which allows to estimate the overall hierarchy consumption as well as to perform an energy breakdown that includes individual values for all of the levels involved in our simulated microarchitecture: data level 1, instruction level 1, level 2 and level 3.

To estimate the energy associated with load instructions we measure, for each level of the cache (that we generically denote as L), the amount of times a data is searched ($L.Lookup$), the amount of hits experienced ($L.Hit$) as well as the number of misses ($L.Miss$). Related with this last parameter, the amount of writings ($L.Delivery$) performed when a required data is not found in the corresponding level is also recorded.

Regarding store instructions energy consumption, as we simulate a microarchitecture with inclusive caches and therefore writing in first level data cache implies writing in upper levels too, we record the amount of stores that consolidate the corresponding data, i.e., the number of committed stores (denoted as *Stores* in our model).

For load instructions, we analyzed two different ways for accessing to tag and data cache arrays: parallel and sequential fashion, although we finally choose parallel access to instruction cache and sequential access to data caches.[34]

We denote the consumption per tag array access involved as $L.r.tag$ and the consumption per data array reading as $L.r.line$. The sum of the two prior values is referred to $L.r$ and the consumption involved in accessing the cache to perform a writing as $L.w$. Considering the previous definitions, the dynamic energy consumption for each L level cache in our model is:

$$Lparallelconsumpt. = L.Lookup * L.r + (L.Delivery + Stores) * L.w \qquad (1)$$

$$Lsequentialconsumpt. = Lookup * L.r.tag + Hits * L.r.line + (L.Delivery + Stores) * L.w\,. \quad (2)$$

In order to clarify this model, Fig. 6 illustrates the memory hierarchy of the simulated microarchitecture where — apart from the three levels of cache — the LSQ, the line fill request queue (LFRQ), the miss buffer (MB) and the main memory are shown too. This figure includes quantitative information about the amount of loads, stores and instruction fetchs that travel for the different structures involved in a 400M instructions simulation of *gcc* application. The *Delivery* parameter is represented with thick arrows going from an upper level to the lower one in order to satisfy the missing data or instruction. The *Stores* parameter is represented with an arrow going from DL1 to L3 (touching also the L2).

Regarding static energy consumption, it is calculated based on the amount of execution cycles reported by the simulation runs.

To complete our energy model, we extract the data shown in Table 4 from CACTI 5.3[26] using 45 nm technology, 360 K temperature, ITRS-HP memory type, a conservative interconnect projection type and semi-global wire outside mat.

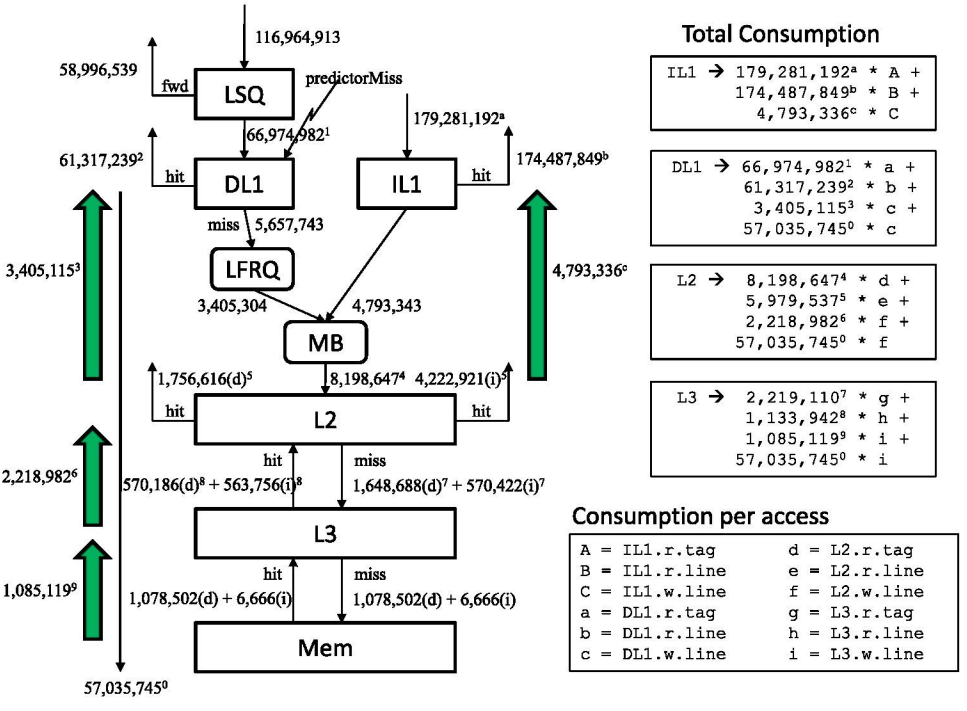

Fig. 6. Energy model applied to *gcc* application case. (*d*) and (*i*) refer to data and instructions, respectively.

Table 4. Energy parameters extracted from CACTI 5.3, being "*L*" the generic level cache.

	IL1 32KB	DL1 16KB		L2 256KB	L3 4MB	
	1 bank	1 bank		1 bank	2 banks	4 banks
	1 port	1 port	2 ports	1 port	1 port	
L.r(nJ)	0.147759	0.141283	0.211438	1.20518	2.59934	3.92063
L.w(nJ)	0.140818	0.135473	0.205607	1.17885	2.5136	3.78945
Leakage(nJ)	0.021913	0.014391	0.046448	0.237133	4.85672	6.292948
L.r.tag(nJ)	0.00401904	0.00331957	0.0023258	0.01084662	0.03509109	0.0259934
L.r.line(nJ)	0.14373995	0.20811842	0.2091121	1.19433338	2.56424891	2.5733466

6. Evaluation

In Sec. 6.1 we detail the results obtained from our DL1 filtering proposal whereas those from simultaneously disabling sets in L2 cache and filtering DL1 accesses are analyzed in Sec. 6.2.

6.1. *Dynamic energy reduction in DL1*

In this section, first we analyze the effectiveness of the studied forwarding predictors and then the main results derived from our first proposed technique.

6.1.1. *Forwarding predictors*

In order to compare the accuracy of the forwarding predictors evaluated — Bloom Filter, Bimodal (with 1 and 2 bits per entry) and Bimodal (2 bits) plus Bloom Filter — we follow Grunwald *et al.* and employ the following metrics used in confidence estimation for speculation control[35]:

Predictive Value of a Positive Test (PVP): It identifies the probability that the prediction of a load as dependent is correct. It is computed as the ratio between the number of correctly dependent-predicted loads and the total number of loads predicted as dependent.

Predictive Value of a Negative Test (PVN): It identifies the probability that the prediction of a load as independent is incorrect. It is computed as the ratio between the number of mispredicted independent loads and the total number of loads predicted as independent.

In our case, using predictors with a high PVP avoids degrading performance. On the other hand, if many loads are incorrectly independent-predicted (high PVN), many cache accesses are carried out unnecessarily, resulting in missed opportunities to reduce the DL1 energy consumption. Therefore, in our design, only very high PVP values are acceptable.

In Fig. 7, we visually present the measurements of PVP and PVN for different sizes in all studied predictors. Intuitively, as we increase the size of any predictor,

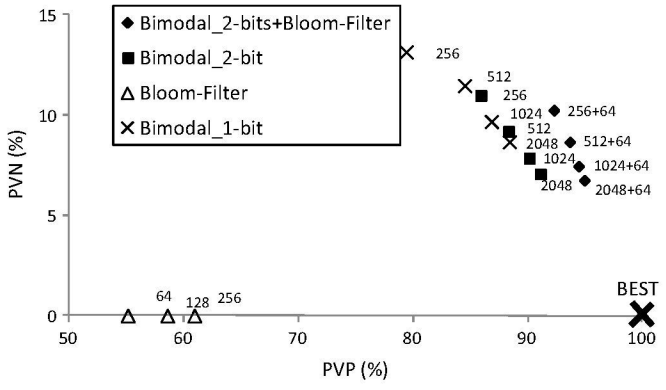

Fig. 7. PVP and PVN values for studied predictors. The results shown are the average values for all applications. For Bimodal Predictors (1 and 2 bits) the data points reflect sizes of 256, 512, 1 K and 2 K. For Bloom Filter we show results for 64, 128 and 256 entries. Finally, the combined predictor uses a 64-entry Bloom Filter and a Bimodal Predictor (2 bits) with 256, 512, 1 K and 2 K entries.

PVP augments and PVN decrease, leading to a better predictor behavior. Note that PVN for Bloom Filter is always zero, since no *false negatives* exist — when a load is independent-predicted, the predictor is never mistaken.

From this figure we can conclude — according to the intuition — that combining the past forwarding information (Bimodal predictor) and memory addresses (Bloom Filter) result in the most accurate predictor (up to 95% of hits for predicted-dependent loads and only around 6% of misses for predicted-independent loads). Thus, we focus on this combined predictor (specifically, we choose a 1K-entry bimodal predictor and a 64-entry Bloom Filter) for thorough analysis reported in next section.

6.1.2. *Main results*

First, we inspect in Fig. 8 the amount of forwarding available in each of the architectures considered (baseline, Nicolaescu's proposal and our architecture). As the

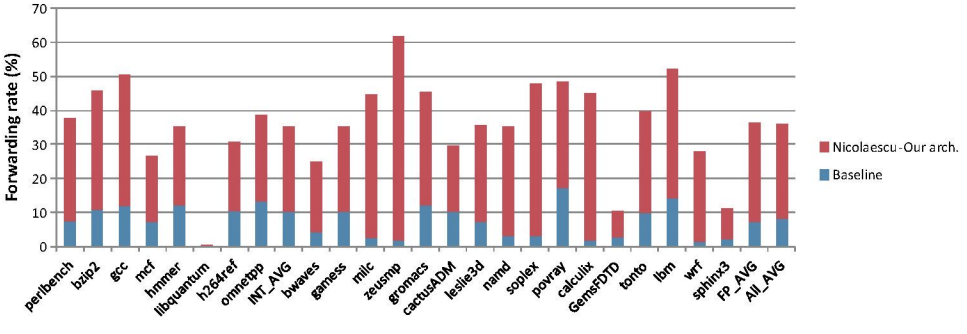

Fig. 8. Forwarding rate: CLSQ contribution.

forwarding rate in both our proposed architecture and that of Nicolaescu is almost exactly the same, in the figure we illustrate the same total rate observed in each application employing these two architectures, as well as the baseline contribution. As shown, the average forwarding rate rises from 8% to 36% for the considered applications when the CLSQ is introduced in the design, so the contribution of Nicolaescu's idea becomes very significant.

Second, a key aspect to be considered is to know how the forwarding rate increment impacts on performance and energy consumption when our proposed DL1 filtering technique is implemented. In Fig. 9 the slowdown observed with respect to the baseline is shown for both Nicolaescu's proposal and ours.

According to Fig. 9, where Nicolaescu's proposal applied to the base architecture drops performance by 1.04% on average (due to the one-cycle delay associated with loads that do not find the corresponding data in the LSQ structure), our prediction mechanism — since no delay occurs in accessing the data cache when a load is predicted as independent — manages to reduce the slowdown to just 0.09% with respect to the baseline.

In order to evaluate the impact of our filtering technique on energy consumption, we track the amount of events detailed in our energy model (*Lookup*, *Hits*, *Miss* and *Delivery* as well as the number of store instructions) for each level cache. In Fig. 10 we show some of these parameters values for both Nicolaescu's proposal and our filtering technique normalized to those obtained in the baseline.

As shown in the figure, the mechanism proposed by Nicolaescu allows to reduce the amount of DL1 accesses by 30% without significantly affecting the remaining levels. With our proposal the reduction is slightly lower, around 26%, due to the contribution of those loads erroneously predicted as independent that unnecessarily access the DL1.

Considering data collected in Figs. 9 and 10, and applying our described energy model, we may obtain the energy reduction derived from our mechanism implementation. In Fig. 11 we show the dynamic energy consumption (mJ) of DL1 for the three microarchitectures under study (baseline, that of Nicolaescu and ours) and

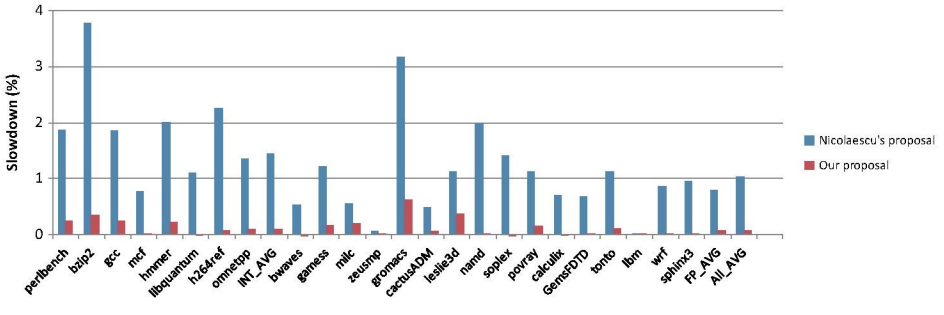

Fig. 9. Performance impact of both our proposed technique and that of Nicolaescu.

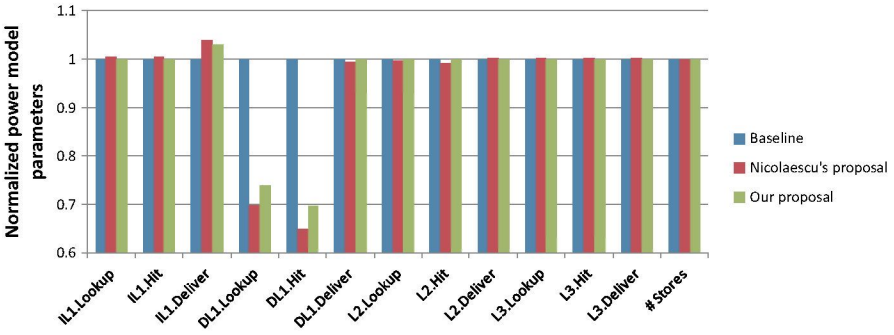

Fig. 10. Energy model parameters for the studied architectures.

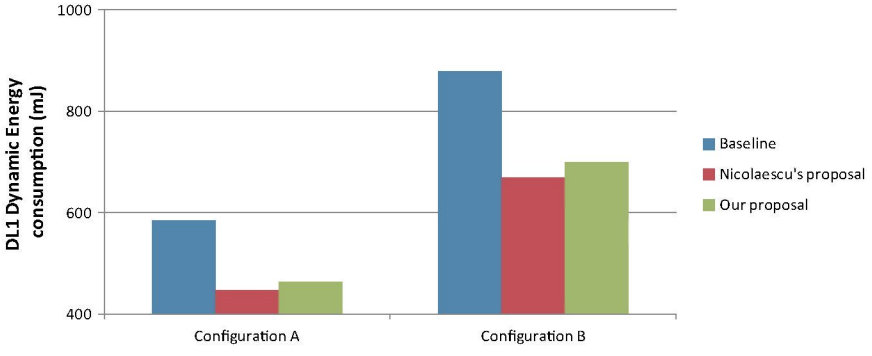

Fig. 11. DL1 dynamic energy consumption for considered proposals.

varying the number of ports in the first level cache. Thus, configuration A is the configuration detailed in the experimental framework section whereas configuration B employs a dual ported DL1 instead of a single one.

From this figure we derive that our filtering proposal manages to reduce the DL1 dynamic energy consumption by around 21.5% compared to the baseline in the two studied configurations. The energy savings employing Nicolaescu's proposal is slightly higher, around 24%, at the expense of a significantly higher slowdown with respect to our proposal as previously shown. Moreover, as a consequence of the reduced slowdown, whereas with our filtering technique the static consumption in the cache hierarchy remains largely unchanged, Nicolaescu's proposal leads to a considerable increment.

6.2. *Static energy reduction in L2*

As established in Sec. 4, we try to reduce the static contribution to cache hierarchy energy consumption by turning off some associative sets in L2. The effect observed

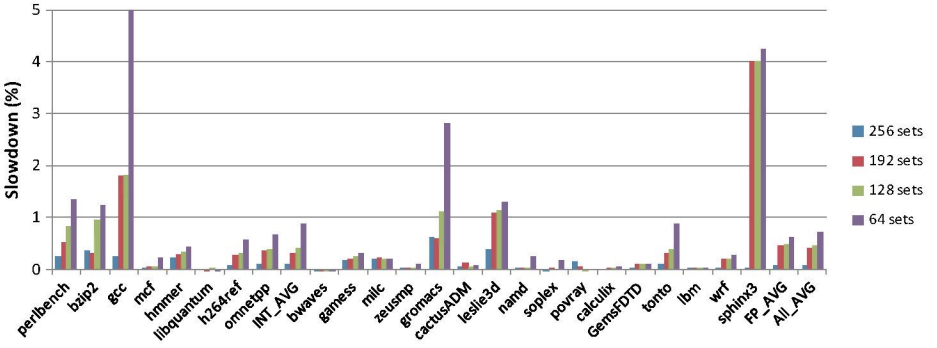

Fig. 12. Performance impact of our combined proposals.

over execution time is shown in Fig. 12, which reports the system slowdown — compared to the baseline — when 256 (full L2 cache), 192, 128 and 64 associative sets are employed. One thing worth noting is that these results are obtained including the filtering DL1 mechanism proposed in Sec. 3.

From the figure above, we extract the average slowdown considering all applications when we turn off different amount of associative sets. Our entire L2 cache (256 sets) in conjunction with our DL1 filtering proposal, according to results shown in the previous subsection too, just drops performance by 0.09% on average compared to the baseline. When just 192, 128 and 64 associative sets are working, this average slowdown rises to 0.41%, 0.45% and 0.72%, respectively.

In Fig. 12 we also zoom into particular applications to analyze individual behavior. Considering the observed slowdown per benchmark when turning off different amount of associative sets, and according to the energy-performance trade-off stated in Sec. 4, we may choose a specific L2 turning off. Recall that for each application we choose the configuration that using the smallest amount of L2 sets — and including our DL1 filtering proposal — reports a slowdown under 1% compared to baseline. Thus, for example, *perlbench* and *hmmer* meet the trade-off using just 128 and 64 sets respectively, whereas for *sphinx*3 all the available sets must be employed to keep performance drop below 1%.

Overall, according to our energy model, by disabling L2 sets in this selective fashion — to cut static L2 energy consumption — and simultaneously applying our DL1 filtering mechanism — that reduces DL1 dynamic energy — the static L2 and the dynamic DL1 contributions to energy consumption drop by 62.2% and 21.5%, respectively on average, leading to a reduction in the combined L1 and L2 total energy consumption of 29.2%, whereas the performance penalty is just 0.25%. As inferred from this reduced slowdown, the increase in L3 accesses due to turning off L2 sets has an almost negligible impact on system performance and even total L3 energy consumption remains largely unchanged (it augments by just 0.3%).

7. Conclusions

In this paper we have proposed a L1 data cache filtering mechanism oriented to reduce the energy consumption. Using the idea of CLSQ to augment the forwarding available in the applications, we manage to cut the dynamic energy consumption of the first level data cache by 21.5% with a negligible slowdown. Furthermore, we also propose to selectively turn off some L2 associative sets in order to reduce the static energy consumption. Combining both techniques a reduction in overall L1 and L2 energy consumption close to 29.2% is achieved, with just a 0.25% performance penalty. In this paper we have focused on just cache energy consumption, since determining the impact of these approaches over the whole processor energy consumption exceeds the goal of this work.

Besides, we have introduced a detailed energy model and a meticulous simulation method to obtain single triggering points for simulation in SPEC CPU2006 applications on an x86-64 architecture.

Acknowledgments

This work has been supported in part by the Spanish government through the research contract CICYT-TIN 2008/508, TIN2012-32180, Consolider Ingenio-2010 CSD2007-0050 and the HIPEAC-3 European Network of Excellence.

References

1. F. Bower, D. Sorin and L. Cox, The impact of dynamically heterogeneous multicore processors on thread scheduling, *IEEE Micro* **28** (2008) 17–25.
2. M. D. Hill and M. R. Marty, Amdahl's law in the multicore era, *IEEE Computer* **41** (2008) 33–38.
3. J. L. Aragon, J. Gonzalez and A. Gonzalez, Power-aware control speculation through selective throttling, *Proc. HPCA* (2003), pp. 103–112.
4. J. Dai and L. Wang, Way-tagged cache: An Energy-efficient L2 cache architecture under write-through policy, *Proc. Int. Symp. Low Power Electronics and Design (ISPLED)* (2009), pp. 159–164.
5. T. V. Kalyan and M. Mutyam, Word-interleaved cache: An energy efficient data cache architecture, *Proc. Int. Symp. Low Power Electronics and Design (ISPLED)* (2008), pp. 265–270.
6. V. Kontorinis, A. Shayan, D. M. Tullsen and R. Kumar, Reducing peak power with a table-driven adaptive processor core, *Proc. 42nd Annual IEEE/ACM International Symp. Microarchitecture (MICRO 42)* (2009), pp. 189–200.
7. IBM Home page, Available at http://researcher.ibm.com/view_project.php?id=1515 (accessed January 2012).
8. M. Monchiero, R. Canal and A. Gonzalez, Power/performance/thermal design-space exploration for multicore architectures, *IEEE Trans. Parallel Distrib. Syst.* **19** (2008) 666–681.

9. Y. Etsion and D. G. Feitelson, L1 cache filtering through random selection of memory references, *Proc. 16th Int. Conf. Parallel Architecture and Compilation Techniques (PACT '07)* (2007), pp. 235–244.
10. D. Nicolaescu, A. Veidenbaum and A. Nicolau, Reducing data cache energy consumption via cached load/store queue, *Proc. Int. Symp. Low Power Electronics and Design* (2003), pp. 252–257.
11. P. Racunas and Y. N. Patt, Partitioned first-level cache design for clustered microarchitectures, *Proc. ICS* (2003), pp. 22–31.
12. J. Kin, M. Gupta and W. Mangione-Smith, The filter cache: An energy efficient memory structure, *Proc. Micro* (1997), pp. 184–193.
13. D. Albonesi, Selective cache ways: On-demand cache resource allocation, *J. Instruction-Level Parallelism* **2** (2000) 1–6.
14. H. Lee, M. Smelyanskiy, C. Newburn and G. Tyson, Stack value file: Custom microarchitecture for the stack, *Proc. HPCA* (2001), pp. 5–14.
15. L. Jin and S. Cho, Reducing cache traffic and energy with macro data load, *Proc. ISLPED* (2006), pp. 147–150.
16. P. Carazo, R. Apolloni, F. Castro, D. Chaver, L. Pinuel and F. Tirado, L1 data cache power reduction using a forwarding predictor, Lecture Notes in Computer Science, Vol. 6448, Springer-Verlag, (2011), pp. 116–125.
17. S. Subramaniam and G. Loh, Store vectors for scalable memory dependence prediction and scheduling, *Proc. HPCA* (2006), pp. 65–76.
18. I. Park, C. Ooi and T. Vijaykumar, Reducing design complexity of the load/store queue, *Proc. of Micro* (2003), pp. 411–422.
19. M. Powell, S. H. Yang, B. Falsafi, K. Roy and T. N. Vijaykumar, Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories, *Proc. Int. Symp. Low Power Electronics and Design (ISPLED)*, Rapallo, Italy (2000), pp. 90–95.
20. S. Kaxiras, Z. Hu and M. Martonosi, Cache decay: Exploiting generational behavior to reduce cache leakage power, *Proc. Int. Symp. Computer Architecture (ISCA)* (2001), pp. 240–251.
21. K. Flautner, N. S. Kim, S. Martin, D. Blaauw and T. Mudge, Drowsy caches: Simple techniques for reducing leakage power, *Proc. Int. Symp. Computer Architecture (ISCA)* (2002), pp. 148–157.
22. F. Castro, D. Chaver, L. Pinuel, M. Prieto, M. Huang and F. Tirado, A load-store queue design based on predictive state filtering, *J. Low Power Electronics* **2** (2006) 27–36.
23. B. Bloom, Space/time trade-offs in hash coding with allowable errors, *Commun. ACM* **13** (1970) 422–426.
24. S. McFarling, Combining branch predictors, Technical report tn-36, Western Research Laboratory, Digital Equipment Corporation (1993).
25. S. Sethumadhavan, R. Desikan, D. Burger, C. Moore and S. Keckler, Scalable hardware memory disambiguation for high ILP processors, *Proc. of IEEE/ACM International Symposium on Microarchitecture* (2003), pp. 399–410,
26. Cacti page at HP labs home page, available at http://www.hpl.hp.com/research/cacti/.
27. M. T. Yourst, PTLsim: A cycle accurate full system x86-64 microarchitectural simulator, *Proc. of ISPASS* (2007), pp. 23–34.
28. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, The microarchitecture of the Pentium 4, *Intel Technol. J.* **5** (2001) 1–13.
29. Copenhagen University College of Engineering, The Microarch of Intel and amd cpu's: An optimization guide for assembly programmers and compiler makers (2009).

30. SPEC 2006 Home page, Available at http://www.spec.org/cpu2006 (accessed November 2011).
31. Gprof home page, Available at http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html (accessed November 2011).
32. Simpoint home page, Available at http://cseweb.ucsd.edu/~calder/simpoint/(accessed November 2011).
33. Xen home page, Available at http://www.xen.org (accessed November 2011).
34. A. González, F. Latorre and G. Magklis, Processor microarchitecture: An implementation perspective, Synthesis Lectures on Computer Architecture, Vol. 5, Morgan & Claypool Publishers (2010), pp. 1–116.
35. D. Grunwald, A. Klauser, S. Manne and A. Pleszkun, Confidence estimation for speculation control, *Proc. of ISCA* (1998), pp. 122–131.