A Unified DVFS-Cache Resizing Framework

Vasileios Spiliopoulos
Uppsala University
vasileios.spiliopoulos@it.uu.se

Andreas Sembrant
Uppsala University
andreas.sembrant@it.uu.se

Georgios Keramidas
Think Silicon S.A.
g.keramidas@think-silicon.com

Erik Hagersten
Uppsala University
erik.hagersten@it.uu.se

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

Abstract

Cache resizing and DVFS are two well-known techniques, employed to reduce leakage and dynamic power consumption respectively. Although extensively studied, these techniques have not been explored in combination. In this work we argue that optimal frequency and cache size are highly affected by each other, therefore should be studied together.

We present a framework that drives DVFS and Cache Resizing decisions in a unified, co-ordinated way. We show that MLP is the key to understand how performance is affected by both techniques and we develop an analytical model to quantify performance variation under different cache sizes and core frequencies. Finally, we expose this information to the OS and/or the application, which are responsible for setting core frequency and cache size based on energy-efficiency policies defined by the user.

Our experimental results show that our model can drive DVFS and Cache Resizing decisions to reduce dynamic and static energy consumption and improve EDP by 18% on average for SPEC2006. We evaluate different policies and showcase that with our model, it is trivial to build any policy involving energy-performance requirements.

1. Introduction

In the last decade, energy has evolved to a first-order constraint, and computer systems are now optimized for energy efficiency. Two of the most important parameters associated with the energy efficiency of a system are the core frequency and the Last Level Cache (LLC) size. Although processors are optimized for 'common case', the optimal cache/frequency configuration for many applications does not lie on the nominal values. Hence, significant effort is expended to identify cases where core frequency or cache size can be scaled down without inordinately affecting performance but yielding energy savings, thus improving the energy efficiency of the system.

In the past, researchers treated DVFS and Cache Resizing as two independent problems, aiming to attack dynamic and leakage energy consumption respectively [6, 13, 25, 19, 22, 36, 37]. This approach, however, disregards the effect of the miss behavior of an application in both techniques. Resizing the cache can turn a compute-bound program to memory-bound, if the cache is not big enough to store the application's working set. At the same time, most DVFS techniques [6, 13, 25, 19, 26] exploit the application's memory slack, i.e., the periods that the processor is waiting for the memory, to scale frequency down and save energy without inordinate performance overhead. In other words, turning the 'cache size' knob can significantly affect the way that the 'DVFS' knob should be controlled. Similarly, the importance of leakage power grows at lower frequency levels, therefore aggressive Cache Resizing can be more beneficial. Moreover, many times the same performance can be achieved by either shrinking the cache or reducing the core frequency, hence the two problems cannot be studied in isolation: we need a unified framework to maximize efficiency by intelligently applying both techniques.

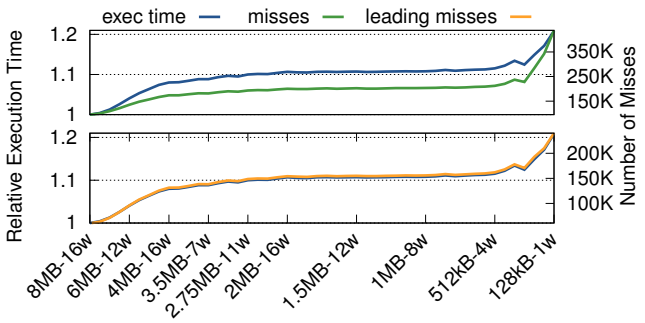

Figure 1: Performance and miss profile for cactusadm under different cache configurations.

Previous approaches used the miss ratio or miss rate to resize the cache [4, 37]. These metrics, however, do not directly reflect how performance is affected by resizing because not all misses are equally important. If shrinking the cache forces hits to turn into misses, but these misses overlap with previously existing misses, the performance penalty is not as significant as it would have been, had these extra misses been isolated.

Figure 1 shows how execution time, number of misses,

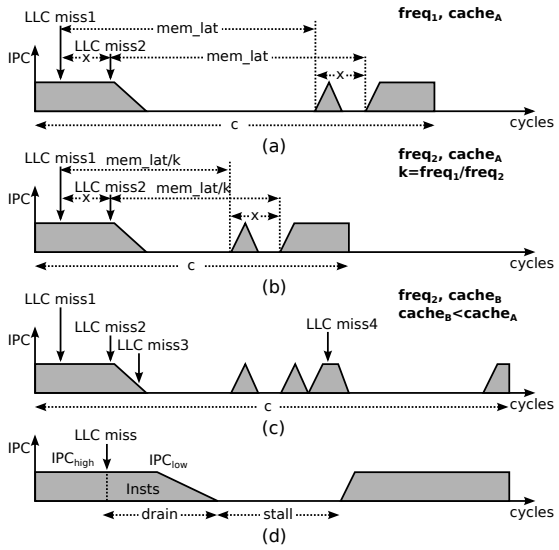

Figure 2: Interval model for DVFS and Cache Resizing.

and number of leading misses change with cache size for *cactusadm*, one of the SPEC2006 benchmarks [10]. The term *leading miss* [6, 13, 26] refers to the first miss in a group of overlapping misses. To compare which of the misses/leading-misses curve correlates better with execution time, each curve is scaled to its maximum dynamic range. The figure shows that execution time does not change proportionally to the change in the number of misses, whereas the leading-misses curve perfectly overlaps with the execution time curve. Hence, it is the leading-misses curve that correlates better with execution time. This motivates us to use this metric for modeling Cache Resizing. More importantly, the same metric has been used to model frequency scaling. This raises the question: can a unified leading-miss model describe both DVFS and Cache Resizing? In this work, we show how to construct such a model and apply it at runtime.

The contributions of this paper are the following:

- We develop a unified DVFS-Cache Resizing model to estimate execution time at any frequency-cache configuration (Section 2).
- We propose a cache-tag architecture to predict cache misses and leading misses from any cache configuration to any other configuration, with low hardware cost (Section 3).
- We feed the model output to the OS to adapt frequency and cache size at runtime, and we showcase why DVFS and Cache Resizing are synergistic (Sections 4 and 5).

2. Performance Modeling

Figure 1 shows that there is a strong correlation between execution time and the number of leading misses of an application. Previous work has also shown that leading misses are the key to model DVFS. In this section, we propose a model to estimate how the execution time of an application changes with cache size and core frequency, by executing it in a single cache and a single frequency. We start from the previously proposed DVFS interval-based model [13, 6, 25] and augment it to account for Cache Resizing. Subsequent sections discuss how to apply the model and evaluate its accuracy.

2.1. Interval-based DVFS model

Interval-based models have been used [11, 7, 8, 9, 33] as a first-order approximation of the execution time of a program. These models assume a steady state IPC, shaped by both hardware (issue width, cache architecture, branch prediction etc.) and software (instruction level parallelism, branch behavior) aspects. Steady-state intervals are punctuated by miss-intervals, introducing stall cycles to the processor. Typically, short-latency miss-events are treated as part of the steady-state intervals and their effect is part of the steady-state IPC. Hence, it is the long-latency events, such as LLC-misses, that introduce stall-cycles to the processor.

Figure 2a demonstrates the interval model with two LLC misses. After the first miss is issued, the processor keeps executing instructions until the miss reaches the head of the ROB, or until there are no more independent instructions in the instruction queue. At this point the processor stalls. After the miss is serviced, the processor starts issuing instructions again until it stalls due to LLC miss2. After the second miss is serviced, the processor can issue instructions again.

In the past, frequency scaling has been modeled using the misses that initiate groups of overlapping LLC misses, named *leading misses* [13, 6, 25]. Other miss-events, such as branch mispredictions or L1 misses, are in-core events and their latencies (expressed in core cycles) are not affected by frequency scaling. In Figure 2a, the second miss is issued x cycles after the first one, hence it will also be serviced x cycles after the first one. From the core's perspective, scaling frequency down by a factor of k is nothing more than scaling the memory latency, expressed in core cycles, by a factor of k, too. Therefore, Figure 2b shows that only the miss interval of the first miss is affected by frequency scaling, whereas the additional cycles to service the second miss remain intact.

If lm is the total number of leading misses in a program, the impact of scaling frequency from $freq_1$ to $freq_2$ ($k = \frac{freq_1}{freq_2}$) is given by the following equation [13]:

$$cycles(freq_2) = cycles(freq_1) + lm \times mem\_lat \times \left(\frac{freq_2}{freq_1} - 1\right) \quad (1)$$

2.2. Interval-based Cache Resizing model

MLP is a key metric to quantify the impact of Cache Resizing on performance, since the cost of a miss overlapping with another miss is dramatically lower than that of an isolated miss. Consequently, a first-order metric of the performance overhead due to a reduced cache size would be to estimate not only the number of extra misses incurred, but also how many of them will not be overlapping with previously existing misses. Figure 2c shows what happens when we reduce the cache size

from $cache_A$ to $cache_B$. Two more misses occur, LLC miss3 and LLC miss4, but only the latter one is a new leading miss: LLC miss3 overlaps with previously existing misses, therefore it does not add significant penalty to the execution time. LLC miss4, on the other hand, introduces a whole new miss interval, thus it is more harmful for performance.

With these observations in mind, a first-order model to estimate the impact of cache resizing in performance can be constructed by assuming that the change in the number of cycles is simply the difference in the number of leading misses, multiplied by the cost incurred by each leading miss. Let $cycles(cache_A)$ be the execution cycles and lm_A the number of leading misses in $cache_A$. The execution cycles in a new configuration $cache_B$ can be estimated as:

$$cycles(cache_B) = cycles(cache_A) + (lm_B - lm_A) \times cost \quad (2)$$

We discuss how to estimate the leading misses in $cache_B$, lm_B, in Section 3.3. Regarding the *cost* of each leading miss, we found that the average memory latency makes an excellent proxy. This seems counter-intuitive, due to the fact that the processor can keep issuing instructions for several cycles after a miss is issued. However, it also takes tens of cycles until the request is propagated through the cache hierarchy to the memory controller, therefore by the time the memory access is issued, the issue window has most likely ran out of independent instructions and the processor has stalled. To verify this assumption, we ran an oracle predictor for the leading misses, leaving the *cost* as the only uncertain part of the model, and we found that this simple proxy yields great accuracy.

2.3. Putting it all together: a Frequency Scaling - Cache Resizing Model

Paying a closer look at the two performance models presented in this section reveals that they are tightly connected. Equation 1 shows that the key to estimate the impact of frequency scaling is the *absolute number of leading misses*. Equation 2 depends on the *difference of leading misses* between cache configurations. Therefore, the two equations can be combined, allowing us to run an application in a base configuration $(cache_A, freq_1)$ and estimate execution time in a target configuration $(cache_B, freq_2)$. This is expressed as

$$\begin{aligned} cycles(cache_B, freq_2) = & cycles(cache_A, freq_1) + \\ & (lm_B - lm_A) \times cost + \\ & lm_B \times mem\_lat \times \left(\frac{freq_2}{freq_1} - 1 \right) \end{aligned} \quad (3)$$

3. Design and Implementation

The performance model of the previous section requires the number of leading misses in any cache configuration as an input, which is not readily available when running an application in a single configuration. In this section we propose a novel method for estimating the number of leading misses in any cache configuration. Then, we show how this information can be fed to the model of Section 2 to estimate performance variation in any cache and frequency, by only running the application in a single configuration. We also discuss heuristics to take into account the effect of hardware prefetching. Finally, we consider implementation details and we prune our model to make it suitable for Pseudo-LRU caches.

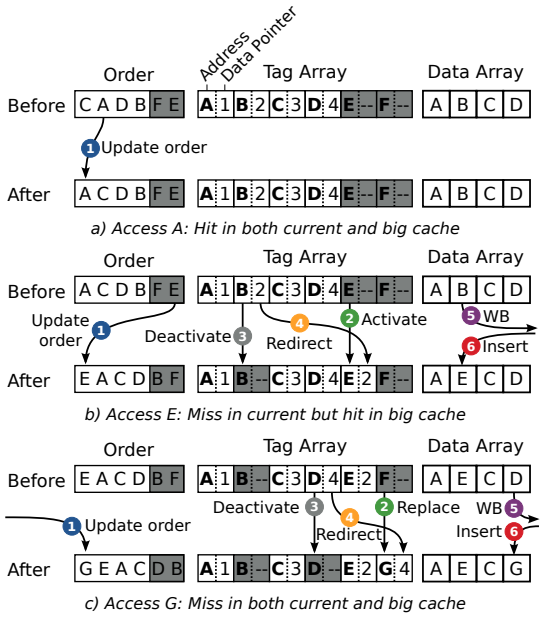

Figure 3: Mechanics to maintain over-provisioned information in the cache tag-array.

Previous work [37, 24] proposed the use of *miss-tags* to track the cache behavior in different associativities. In this approach, an extra tag-array is used to keep track of the blocks that would be in a cache of a different associativity. Qureshi et. al. [24] observed that for LRU caches a single miss-tag-array using the maximum associativity of interest is sufficient, and the LRU-order can be used to determine behavior in smaller associativities. However, they still pay the overhead of introducing an extra tag array. In this section, we demonstrate a novel method for integrating the miss-tags into the regular tag-array. Furthermore, previous work have disregarded the effect of the hardware prefetcher into the miss-ratios in different caches: miss-tags can overestimate the number of misses, if certain accesses that would miss in small caches could be turned into hits by the prefetcher. Here we overcome this limitation by characterizing every access as *prefetchable* or not. Finally, previous work have not estimated the overlapping of misses in different configurations, which is the key insight of our performance model proposed in Section 2.

3.1. Modeling Misses in Different Associativities

Figure 3 shows the mechanics for maintaining over-provisioned tag information in an example 4-way cache, that can be upsized up to 6 ways. Shaded entries indicate inactive

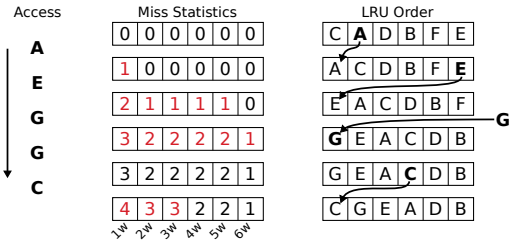

Figure 4: Miss counters for estimating number of misses in different cache associativities.

tags. To determine cache behavior in different cache sizes, we only need tag and LRU information for excessive addresses, therefore the data array does not contain inactive parts. In the course of this example, it will become obvious that, since tag and data arrays are not of the same size anymore, there is no one-to-one mapping between their entries. Therefore, for each cache line, along with the tag we need a pointer to the corresponding entry in the data array.

Figure 3a shows the case that an access hits in both the current, 4-way cache, and the big, 6-way cache. In this case, we only need to update the LRU-order (*A* is now the MRU block). In Figure 3b, accessing address *E* misses in the current cache but it would hit in the big cache, since an inactive entry exists in the tag array. In the current cache, we should evict the least recently used cache line, which is *B*, but since *B* would still be present in the big cache, we simply deactivate it. On the other hand, we need to bring in *E*, which is already present in the tag array as an inactive tag, therefore we simply need to mark it as active. The question now is where in the data array should block *E* be placed. Since in the current cache it is block *B* that should be replaced, we evict it from the data array and bring in *E* instead, and we update the data pointer of block *E* in the tag array. We also update the LRU-order, which now indicates E as the most recently used cache line.

Finally, in Figure 3c we access block *G*, which is a miss in both the current and the big cache. Therefore, we do need to replace an entry in the tag array, but this entry is different for the current and the big cache. For the big cache, block *F* is the LRU block, therefore we replace it with block *G*. In the current cache, on the other hand, block *D* is the LRU one, therefore we deactivate it (but keep it in the tag-array because it would be present in the big cache). We copy the pointer of the deactivated block (*D*) to the pointer of the freshly inserted block (*G*) and we use this pointer to determine the block to be replaced in the data array. Finally, we update the LRU order.

In the example above we described a mechanism to keep track of the LRU-order of the blocks contained in the big cache without affecting the functionality of the current cache. Using this LRU order, we can determine if an access to a block would be a hit or a miss in different associativities: accessing the $n-th$ most recently used block would be a hit in a cache with n ways or more, and a miss otherwise. Of course, any access to a block that does not exist in either the active or inactive part of the tag array is also a miss. Hence, we can characterize every cache access as *hit* or *miss* in different cache configurations, and use counters to indicate the miss counts. Figure 4 shows how these statistics are collected at runtime. Accessing *A* would result in a miss in a 1-way cache and in a hit if $associativity \geq 2$. The next access is on block *E*, which is the least recently used in the cache, therefore it would be a miss unless all 6-ways are enabled. Block *G* is not present in the tags, therefore accessing it results always in a miss, but re-accessing it right away would hit on any cache. Finally, accessing *C* is a hit with 4 or more ways available.

An alternative approach to resize the cache is to change the number of sets. Set-resizing forces blocks that used to be mapped in different sets to content for the same set. To model set-resizing, we keep combined LRU-information for blocks belonging to sets that can conflict with each other under a certain set-resizing configuration. We evaluated this approach and we achieved accuracy similar to the one achieved for way-resizing. However, applying set-resizing on top of way-resizing only marginally improved the efficiency of our policies (Section 5). Therefore, for simplicity, we do not consider set-resizing for the remainder of this paper.

3.2. Modeling Hardware Prefetching

Using miss-tags to estimate mises disregards the effect of hardware prefetcher. This can lead to overestimating misses when predicting from big to small caches, since some of the accesses not fitting in small caches can be brought in the cache by the prefetcher in advance. Hence, we need to identify if an access that would miss in some cache configuration is *prefetchable*, i.e., if it is part of an access pattern the prefetcher is aware of. To the best of our knowledge, this is the first work modeling the effect of the prefetcher in miss-curve estimation.

We consider a stride prefetcher, but similar methods can be applied for different types of prefetchers. We include a prefetch-history buffer (512 entries), which keeps the latest prefetches that would be issued in the minimum-sized cache. In each entry, a cycle-counter indicates if the prefetch was in-time or not. The buffer is direct-mapped, hence the lookup cost is minimal. On every LLC access, we consult the prefetcher to determine if this access would have been prefetched in case of a miss. If an access is found to be prefetchable, it will always be accounted as a hit, no matter what our cache model predicts. The hardware cost for this structure is 40 bits per entry (block address + counter), resulting in 2.5KB in total, which is negligible for an 8MB LLC.

3.3. Modeling MLP

Previous works have proposed techniques to estimate miss-rate curves in different caches [17, 24, 1, 37]. We already discussed how we apply a similar approach for miss estimation, without the overhead of using extra miss-tags. However, miss estimation only roughly indicates the impact of cache size on performance. To build an accurate analytical model, we

need to know how MLP changes with cache size. In our proposed model, we account for MLP through the number of leading misses. Previous work [6, 13, 26] has shown that leading misses are approximately the same across frequencies, therefore we only need to investigate how leading misses change with cache size.

We have shown how to characterize every LLC access as hit or miss in different caches. If an access is predicted to be a miss in some cache, we also need to determine whether this miss overlaps with other misses, i.e., if the miss is a leading miss or not. To do so, we keep a flag and a counter for every possible cache configuration. This only adds minimal hardware cost: if, for example, 16 cache configurations are supported, we only need 16 counters and 16 1-bit flags. The flag indicates a pending leading miss in the corresponding cache configuration, and the counter specifies when the flag should be reset. If an access is characterized as a miss and the flag is not set for a given cache, we set the flag to 1 and increment the projected number of leading misses (for the specific cache) by 1. We also set the counter to the number of cycles, after which the flag should be reset. Subsequent misses in this configuration are assumed to be overlapping with the leading miss. The question is when should the flag be reset, i.e., after how many cycles is the leading miss considered serviced. We use the following heuristic:

- If an access, predicted to be a leading miss, misses in the current cache, the counter is set to average memory latency.
- If an access, predicted to be a leading miss, hits in the current cache, the counter is set to $\frac{Insts}{IPC_{high}}$ (Figure 2d).

For the first case our approximation is fairly obvious: if an access misses in the current cache, every access coming in the next mem_lat cycles overlaps with the leading miss. For the latter case, however, the overlapping of subsequent accesses is not straightforward. The approximation we use is based on our experience from other misses in the current cache. We use the average number of instructions issued under a leading miss, and estimate the number of cycles it would take to issue them under the steady-state IPC. The projected number of cycles is stored in the counter for the specific cache configuration, and the counter decrements on every core cycle. Once the counter reaches 0, the flag is reset and the latest leading miss is now considered serviced. Therefore, the next access estimated to be a miss will be counted as a new leading miss.

3.4. Estimating Performance

Figure 5 shows how our model estimates the number of misses and leading misses from any cache to any other cache (top left and right graphs) for *sphinx3*. Dark blue line indicates the reference number of misses/leading misses. For each cache configuration, i.e., each point on the dark blue line, a light blue line estimates the cache behavior in the whole spectrum of cache configurations. Hence, the difference between dark and light blue lines represents the error in estimating cache behavior from any cache to any other cache configuration.

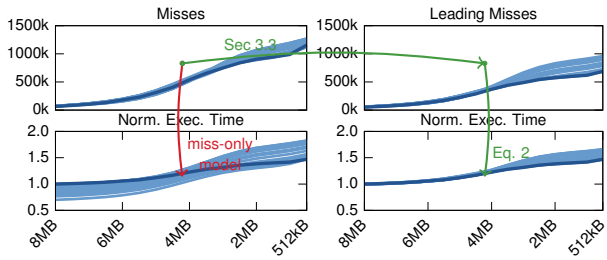

Figure 5: Performance and miss profile for *sphinx3* in different caches. Dark blue lines represent reference, light blue lines represent model estimation. Top graphs show number of misses (left) and leading misses (right). Bottom graphs show execution time prediction based on the number of misses (left) and leading misses (right).

The figure shows that estimating leading misses is less accurate than estimating the total number of misses. This is because, as discussed above, accesses that overlap under the base cache configuration do not necessarily overlap in other cache configurations and vice versa, and the heuristics used to determine the overlapping introduce errors. One could argue that, since estimating total number of misses is more accurate, a performance model should be based on this metric. The bottom left graph shows the accuracy in predicting execution time, if the total number of misses and their corresponding costs are used, similarly to the performance model of Equation 2. As the figure shows, execution time tends to be overestimated at small cache sizes and underestimated in big cache sizes. Paying a closer attention to the misses/leading-misses graphs shows that, in big caches the ratio of misses to leading misses is about 1.25:1, i.e., 4 out of 5 misses are isolated. In small cache sizes, however, more misses overlap with the same leading miss, and this ratio rises to 1.67:1. Therefore, the average cost per miss is much higher in big caches, since almost every miss comes with a whole miss interval of its own, and assuming that additional misses that will occur in small caches will have the same cost overestimates execution time. Similarly, going from small to big caches, misses turn into hits, but only the ones that are not part of a group of misses will have a big impact on execution time. Assuming that all of the misses contribute equally to performance leads to overestimating the benefit of increasing the cache. This shows why simply estimating miss-rate curves [17, 24, 1, 37] is insufficient for performance modeling. In contrast, with our proposed performance model, even if there is some error introduced when estimating leading misses, we still get great performance estimation accuracy across different cache configurations (bottom right figure). This proves that MLP is indispensable for modeling cache resizing, and leads to great prediction accuracy even if there is some uncertainty in its estimation.

Figure 6 shows the accuracy of our model for the SPEC2006 benchmark suite, using the experimental setup described in Section 5. We execute each application in 16 different cache configurations and 2 frequencies, 1.6GHz (green line) and

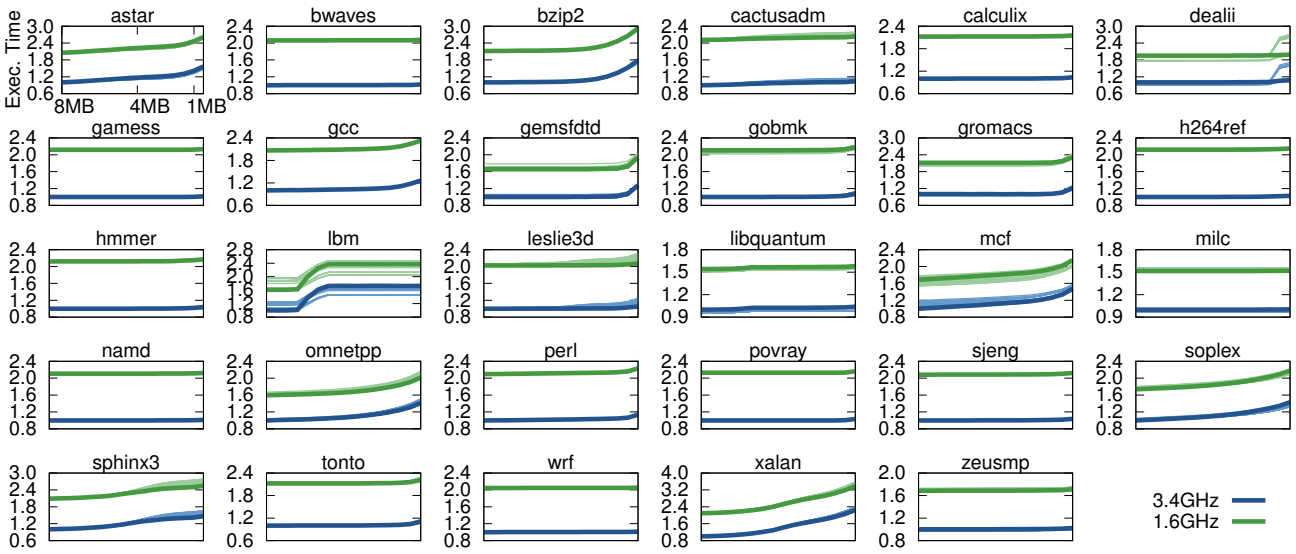

Figure 6: Normalized execution time for different cache sizes and core frequencies. Blue-green lines refer to 3.4GHz-1.6GHz respectively. x-axis represents different caches, ordered from big to small configurations. For every possible cache size and core frequency, one light blue and one light green line estimate execution time for all possible frequency-cache configurations.

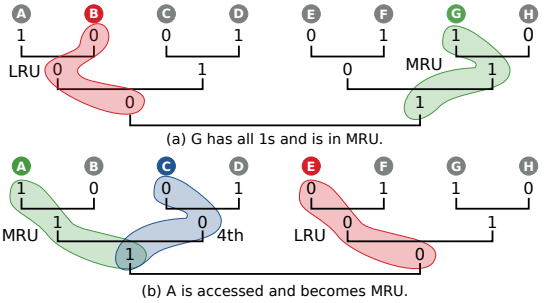

Figure 7: Tree-LRU structure for Pseudo-LRU policy.

3.4GHz (blue line). Due to space limitations, for multi-input benchmarks we show the aggregate results for all inputs. From each run, we estimate execution time in any cache and frequency. The figure shows that most of the times, we achieve great accuracy, especially for a runtime model that needs to capture the relative performance between configurations. The main source of error comes from estimating MLP (*mcf*, *lbm*) and/or prefetcher behavior (*dealii*) between configurations that differ considerably in misses. However, the runtime evaluation will show that our model is good enough to efficiently drive energy-efficiency policies (Figure 10).

3.5. Pseudo-LRU Caches

Because true LRU is expensive, cheaper variations of LRU can be used, called *pseudo-LRU*. The simplest pseudo-LRU replacement policy uses one bit, indicating if a cache line is not the LRU one. When a cache line is accessed, its pseudo-LRU bit is set to 1. Once all the pseudo-LRU bits in a set are 1, they get reset. Once a block has to be evicted, one of the blocks with a pseudo-LRU bit of 0 is picked for replacement.

However, this proxy cannot determine the specific *order* of each cache line. In a 16-way cache, for example, with half pseudo-LRU bits set to 0, the LRU-order of each of them can be anything between 0 and 7, with 0 denoting the LRU cache line. Our cache model needs the LRU-order of each cache line accessed, therefore 1-bit LRU information is insufficient to characterize every access as hit/miss in other configurations.

We use the tree-LRU structure of Figure 7. For every pair of cache lines, 1 bit indicates whether the left/right cache line is the most recently used one in the pair. For the sake of clarity, we use a slightly different representation, with 1 bit per cache line denoting that this line is the MRU (1) or LRU(0) in the pair. Similarly, at each next level of the tree, we order pairs of the previous level. When a block is accessed, the corresponding pair at each level of the tree becomes the most recently used pair. A simple proxy to estimate the order of a block is obtained by walking the tree from the root to the block of interest. In Figure 7b, block A, for instance, has an LRU order of 111 (MRU), whereas block C has an order of 100 (4-th most recently used). The limitation of this approach is that for big tree structures it introduces errors due to limited information for blocks that are too far from each other in the tree. Consider the scenario that only two blocks in the set are accessed, A and E. At any point in time, the most recently used between the two has an order of 111, which is the true order, but the other block's order is 011 instead of 110. This stems from the inherent assumption of a tree-LRU structure that on each sub-tree, all blocks in the MRU half were used more recently than all blocks in the other half of the tree. One way to overcome this limitation is to form several trees, with different block combinations on each of them, to reduce the worst-case distance between blocks, and use the information from all the trees to extract an LRU-order for each block. Due to space limitations, we do not show an exhaustive analysis on this optimization problem. We found, however, that two tree structures estimate LRU-order with reasonable accuracy.

3.6. Implementation Cost

The heuristics discussed take strongly into account implementation cost, in a way that our method can be applied in real systems. For the tag-array extensions, we need one bit per cache line to show if it is active or not. Assuming a $w-way$ cache, we also need $log_2 w$ bits to point where in the data array each block is located. For the tree-LRU structure, we need $\frac{w}{2}$ bits for the first level of the tree, and each next level needs half the bits of the previous level, hence the total number of bits for each set is $\frac{w}{2}+\frac{w}{4}+...+1$. In an example 16-way cache, the total number of extra bits per set is 95. Given that each set consists of 16 64-byte blocks, along with tag-array information (tag address, coherence state, replacement information etc.) the total overhead is about 12% of the tag-array and less than 1% of the total cache. To estimate MLP, for each cache configuration, one bit indicates a pending miss and an 8-bit counter indicates when the flag should be reset (8 bits represent memory latencies of up to 256 core cycles).

Regarding dynamic energy consumption, we are conservative and assume that, even when LLC is downsized, access energy is that of a maximum-sized cache. The dynamic energy will be higher than that of a conventional cache, due to extra operations (determining blocks to be replaced in current/maximum cache, updating data-pointers). However, LLC dynamic power accounts for a small part of total power consumption (6% for Xeon Tulsa processor [15]). Moreover, most of the dynamic power comes from data-movement which is not affected by our extensions. Hence, our cache extensions only marginally increase dynamic energy consumption.

To expose the estimated cache behavior to the OS, we extend the available set of processor performance counters to measure number of misses and leading misses predicted for different configurations. In this work we consider 16 cache configurations, however, a more coarse-grained resizing granularity can be adopted to reduce complexity.

3.7. Summary

We present a methodology to estimate how the number of misses in a cache, and their overlapping, changes with respect to cache size. We propose an asymmetric tag-array and data-array structure, where the tag-array always keeps track of the blocks that would be in the maximum-sized cache by combining the active and inactive tags. At the same time, the active tags keep track of the data in the current cache to preserve traditional cache functionality. We take the hardware prefetcher into account and show that we can efficiently estimate performance across different cache/frequency configurations.

4. DVFS-Cache Resizing Runtime Framework

Having shown that our model tracks performance in different frequencies and cache sizes, we apply the model to control the two knobs at runtime. To estimate energy in different caches/frequencies, we use a previously proposed correlation

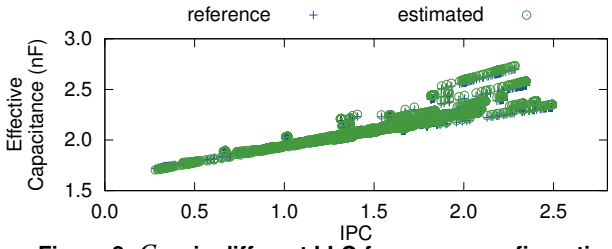

Figure 8: C_{eff} in different LLC-frequency configurations.

model. The advantage of analytical models over empirical methods is the full control of the behavior of the application: instead of blindly fine-tuning empirical parameters and hope for the best, we set the energy and performance requirements and our models satisfy them in an optimal way.

4.1. Energy Estimation

Estimating *effective capacitance* (C_{eff}), i.e., the portion of the total capacitance of the chip that, on average, switches state on each cycle, is the key to model energy consumption [27, 28]. Previous work has shown that there is a strong correlation between C_{eff} and various processor event rates: simpler models correlate C_{eff} with IPC only, whereas more detailed models take into account more events to improve accuracy. The advantage of this method is that it decouples the coefficients of the model from the frequency the model was trained in: the same model can estimate C_{eff} in any frequency, if the event rates collected are properly adjusted for that frequency. Once C_{eff} is estimated, power is simply $P = f \times C_{eff} \times V^2$.

We follow a similar approach to create a power model for our modeled system. We find that a correlation model based on the rate of total instructions (IPC), as well as floating point and memory instructions, yields good accuracy to estimate C_{eff} (Figure 8). Based on this model, the total energy consumption can be estimated by the following equation[1]:

$$\begin{aligned} Power =& f \times C_{eff}(IPC, FPIPC, MIPC) \times V^2 + \\ & core\_static(V) + LLC\_static(V, size) \\ Energy =& Power * t_{exec} + \\ & mem\_accesses \times mem\_energy\_per\_access \end{aligned} \quad (4)$$

Event rates and execution time are estimated using the performance model. Core and LLC static power are known in different configurations. To account for energy consumed by memory, we multiply the number of memory accesses (estimated by our model) with the per-access energy cost.

4.2. Multi-core extensions

Since we target LLC resizing, it is important to extend our methodology for the case that cores running different applications share the same LLC. Tracking misses with our extended-tags architecture is not affected by the presence of more than

[1] IPC: instructions per cycle, FPIPC: FP instructions per cycle, MIPC: memory instructions per cycle

Parameter	Value
Number of Cores	4
Width/ROB/IQ/LQ/SQ/Regs.	8/192/64/32/32/256
L1 Instruction / Data Caches	32kB, 64B, 8-way
L2 cache	128kB, 64B, 8-way
L3 cache	8MB-128kB, 64B, 16-way
Frequency range	3.4GHz-1.6GHz (16 frequency levels)
Voltage range	1.15V-0.97V
DRAM	8GB, 60ns, 12.8GB/s

Table 1: System Configuration. We use the default gem5 out-of-order CPU model. V-f ranges are retrieved from a real Intel Sandybridge machine.

one cores: the access stream in LLC would consist from accesses coming from all cores, hence our miss-estimation takes the effects of sharing LLC into account. The only extension required is to map the leading misses to the cores that issued them, hence we need per-core leading-miss counters and flags (the cost of which is minimal as discussed in Section 3.6). Then, our model can be applied for each core individually to estimate performance and energy of each application in every possible configuration, and pick the configuration that optimizes the system's efficiency.

4.3. Energy-Efficiency Policies

The policies we run aim to find the best balance between performance and energy consumption. To achieve this, we use Energy Delay Product (EDP) as the metric to optimize. Using our performance/energy models, we estimate execution time and energy at any frequency/cache configuration, hence it is straightforward to estimate the optimal-EDP setting. To showcase the importance of a co-ordinated management, we also evaluate policies that apply the two techniques independently.

For every policy, we run an application for a time window and we collect the input required by our model. At the end of the window we run the performance and energy models to estimate the configuration that optimizes the metric of interest. We use a last-value predictor, i.e., we assume that the behavior of the next window will be similar to the current one, hence we apply the setting that is estimated to be the optimal for the current window. Investigating more sophisticated phase-detection schemes is beyond the scope of this paper and is left for future work. We envision our framework to be part of the OS, hence we use a window of 34M instructions[2].

In a multi-core, fully-loaded (all cores active) system, individual applications do not execute the same number of instructions across different scenarios, hence using latency and energy as metrics is inapplicable. Instead, we use the throughput-equivalent metrics. We express the per-application slowdown in terms of Billion Instructions Per Second (BIPS), and the system slowdown is simply the average of them. To express efficiency, we use $avg\_slowdown \times total\_power$, which is equivalent to EDP, but expressed to the unit of time, hence for simplicity we use the term 'EDP' for our efficiency metric.

[2]during an OS-quantum (10msec), a 1-IPC application running at 3.4GHz executes 34M instructions.

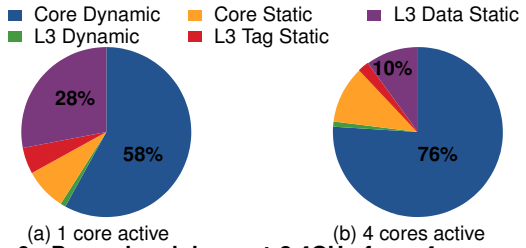

Figure 9: Power breakdown at 3.4GHz for a 4-core machine with (a) one core active and (b) all cores active.

5. Evaluation

We use gem5 [2] to evaluate our framework, with the configuration of Table 1. We extend the caches to support our modeling methodology, and we enhance it with a runtime mechanism that implements our model by collecting the statistics specified in Section 2. We run the whole SPEC2006 benchmark suite, with all different inputs for each benchmark. Due to space limitations, for benchmarks with multiple inputs we merge the results. For each benchmark input, we use 10 different checkpoints taken across its execution, to capture the behavior in different application-phases. Hence, in total we evaluate 550 checkpoints. For each checkpoint, we warm-up the caches for 300M instructions, and then run 500M instructions for the runtime policies. For the profiling runs (Figure 6), due to the huge design space explored, we limit the detailed simulation to 30M instructions. To model energy, we combine McPAT [15] with real-hardware experiments: we run micro-benchmarks on an Intel SandyBridge machine to determine the core and LLC static power. For dynamic power, we found that McPAT does not sufficiently model clock power [34, 21]. Hence, we scale McPAT parameters to fit the ones reported in previous work [3, 14]. Figure 9 summarizes our power-modeling assumptions, showing the average power-breakdown for our processor.

5.1. Adaptive DVFS-Cache Resizing

In single-core experiments, 3 out of 4 cores are power-gated, hence LLC static power dominates. As Figure 9a shows, 28% of total power comes from LLC data-array static power, which indicates a great potential for LLC Resizing. On the other hand, dynamic power is 58%, hence we expect DVFS to be rather conservative. An additional reason for this is that, with an 8MB LLC, most SPEC benchmarks are CPU-intensive.

Figure 10a shows EDP for three different policies, normalized to the EDP of the maximum frequency and cache size. By combining our performance/energy models it is trivial to build similar policies, that can be applied by the OS or the application itself. By only applying DVFS (blue bar), EDP of memory bound applications (e.g. *libquantum*, *milc*) is improved by up to 22%. Many CPU-bound applications, however, such as *calculix* and *h264ref*, are too frequency-sensitive to benefit from DVFS. The Resize Only policy (green bar), on the other hand, achieves significant EDP savings in most of the benchmarks,

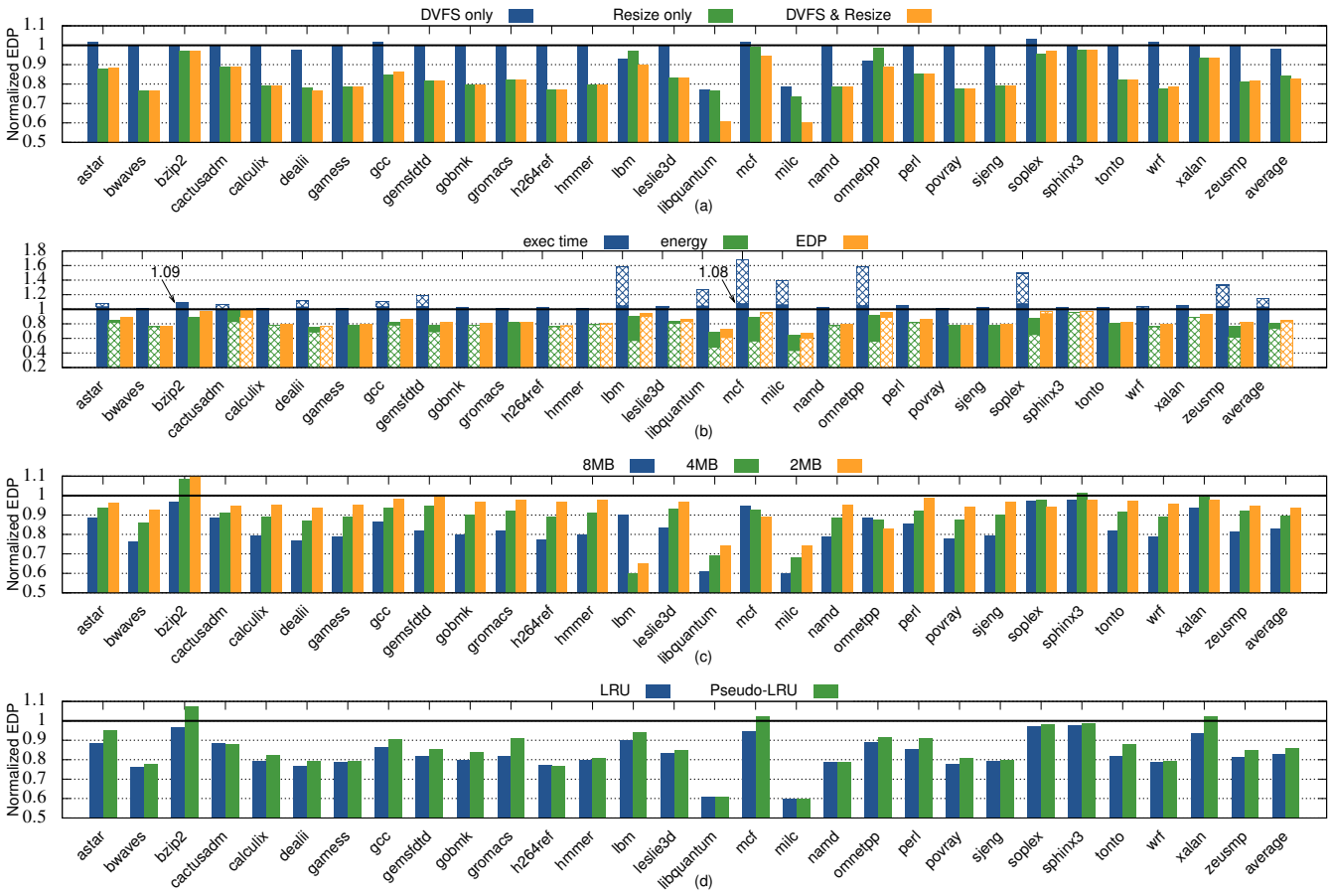

Figure 10: Evaluation for different runtime policies. (a) EDP for three different policies: DVFS only, Resize only and combined DVFS & Resize. (b) execution time, energy and EDP for two variations of the DVFS and Resize policy: one trying to minimize EDP with no limitations (patterned bars), and one minimizing EDP within a 5% performance constraint (solid bars). (c) EDP for the DVFS and Resize policy when the baseline last-level cache is 8MB (blue bars), 4MB (green bars) or 2MB (yellow bars). (d) EDP for LRU and Pseudo-LRU replacement policies.

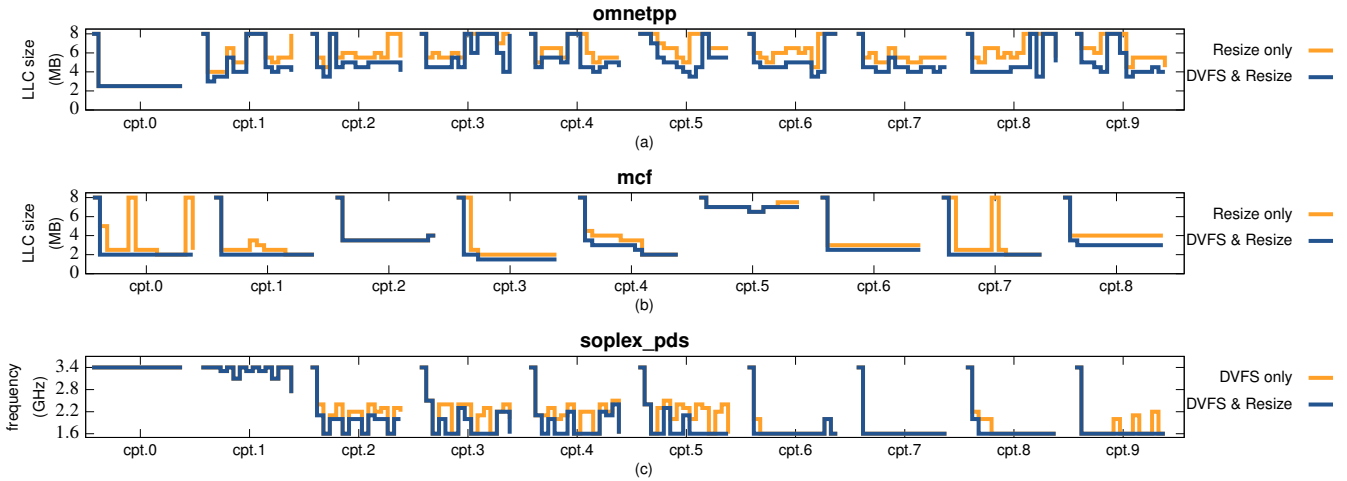

Figure 11: Cache Resizing and DVFS transitions over time for different policies. (a) and (b) compare Cache Resizing decisions for *Resize only* and *DVFS & Resize* in *omnetpp* and *mcf* respectively. In these two applications, resizing the LLC costs performance (Figure 6), hence when DVFS reduces dynamic power, aggressive LLC resizing becomes more beneficial. (c) shows that reducing LLC size leads to more aggressive DVFS decisions due to the reduced static power, hence *DVFS & Resize policy* scales frequency to lower levels compared to *DVFS only* policy in *soplex_pds*.

	app1	app2	app3	app4		app1	app2	app3	app4
mix1	astar$_{(cs,fs)}$	astar$_{(cs,fs)}$	bzip2$_{(cs,fs)}$	gcc$_{(cs,fs)}$	**mix5**	astar$_{(cs,fs)}$	milc$_{(ci,fi)}$	libquantum$_{(ci,fi)}$	soplex$_{(cs,fi)}$
mix2	gamess$_{(ci,fs)}$	hmmer$_{(ci,fs)}$	namd$_{(ci,fs)}$	sjeng$_{(ci,fs)}$	**mix6**	milc$_{(ci,fi)}$	libquantum$_{(ci,fi)}$	mcf$_{(cs,fi)}$	omnetpp$_{(cs,fi)}$
mix3	gemsfdtd$_{(cs,fi)}$	omnetpp$_{(cs,fi)}$	mcf$_{(cs,fi)}$	soplex$_{(cs,fi)}$	**mix7**	milc$_{(ci,fi)}$	namd$_{(ci,fs)}$	mcf$_{(cs,fi)}$	sphinx3$_{(cs,fs)}$
mix4	milc$_{(ci,fi)}$	milc$_{(ci,fi)}$	libquantum$_{(ci,fi)}$	lbm$_{(ci,fi)}$	**mix8**	milc$_{(ci,fi)}$	namd$_{(ci,fs)}$	omnetpp$_{(cs,fi)}$	sphinx3$_{(cs,fs)}$

Table 2: Applications used for multi-workload mixes. Each application is characterized as cache sensitive/insensitive (cs/ci) and frequency sensitive/insensitive (fs/fi), based on Figure 6.

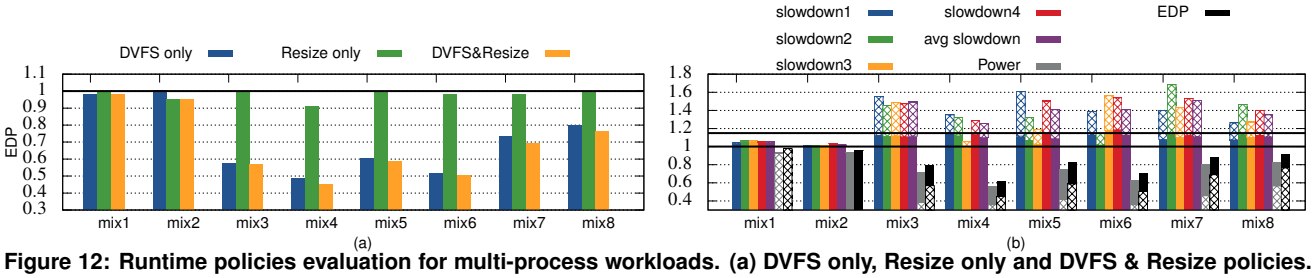

Figure 12: Runtime policies evaluation for multi-process workloads. (a) DVFS only, Resize only and DVFS & Resize policies. (b) Two variations of the DVFS & Resize policy: minimizing EDP (patterned bars) and reducing EDP without harming performance by more than 15% (solid bars). The 15% performance threshold for the conservative policy is indicated by a horizontal line in (b).

since it is rare that the whole 8MB LLC is utilized (also shown in Figure 6). However, the full EDP benefit is only obtained after applying both techniques (yellow bar). This way, our model detects the combined trade-off of DVFS-Cache Resizing and apply it to boost EDP. For cases like *libquantum* and *milc* one can claim that combining the two techniques simply delivers the combination of the savings each of them would deliver in isolation. For cases such as *mcf* and *omnetpp*, however, it is clear that decisions about one technique also affect decisions about the other one. Especially in the case of *mcf*, each of the techniques in isolation cannot improve EDP by more than 1%, but the combination of them gives a 6% benefit. This is because with a reduced LLC static power consumption (imposed by Cache Resizing), DVFS can be more aggressive and trade performance for energy savings. At the same time, reducing dynamic power consumption of the total processor raises the importance of LLC static power, hence Cache Resizing can be more aggressive. In other words, DVFS and Resizing decisions are not orthogonal. This is also shown in Figure 11. The *Resize only* policy is much more conservative for *omnetpp* (Figure 11a), therefore it does not improve EDP. When DVFS is enabled, however, LLC static power becomes more important and *DVFS & Resize* policy takes more aggressive resizing decisions and improves EDP compared to *Resize only* and *DVFS only* policies. The same applies for the resizing decisions of *mcf* (Figure 11b). Finally, Figure 11c shows that Cache Resizing leads to more aggressive DVFS for *soplex_pds*, compared to *DVFS only* policy.

Figure 10b shows the *DVFS & Resize* policy of Figure 10a with the addition of how execution time and energy are affected. Moreover, we show two different variations of that policy: one that tries to maximize EDP savings (patterned bars) and one that tries to achieve the best EDP savings within 5% of performance overhead (solid bars). Since our runtime policies are based on models that accurately capture the energy and performance impact of our decisions, such targeted policies are easy to apply and do not rely on empirical parameters and application-specific fine-tuning. As expected, the aggressive policy (patterned bars) suffers from higher execution time overheads, but at the same time achieves better energy and EDP savings (e.g., *omnetpp*, *libquantum*). For many cache-insensitive applications (e.g., *h264ref*, *gobmk*) the two policies are identical, since frequency is kept at maximum and cache size can be minimized at no cost. There are also cases like *lbm* where the two policies take different decisions but they end up in similar EDP savings. For the conservative policy, execution time is kept within the 5% cap in most cases, with *bzip2* (9%) and *mcf* (8%) being the most notable exceptions.

We also show (Figure 10c) that our method is beneficial in a more constrained environment with a 4MB (green bars) or 2MB (yellow bars) LLC. With a smaller LLC, the energy savings due to resizing are significantly reduced. Also, we expect that LLC size becomes more critical, hence the potential to resize LLC without harming performance is limited. Nevertheless, we achieve significant savings even for a 2MB LLC. In some cases (e.g., *lbm*), the benefit is actually enhanced for a smaller baseline LLC. This can be explained by Figure 6: for certain applications, there is a critical cache size below which further reducing the cache does not affect execution time. If the baseline cache is above that critical size, we cannot apply aggressive cache resizing without affecting performance. But if our baseline cache is smaller than the critical cache size, then the cache can be downsized without any performance overhead. Finally, Figure 10d shows that EDP savings in a Pseudo-LRU LLC are comparable to the LRU-case.

5.2. Multi-process Workloads

Figure 12a shows the results for 8 workload mixes (Table 2). We show how performance (per-application and average), power consumption and EDP are affected. Our workload selection exhibits a variety of cache/frequency sensitive/insensitive applications. Contrary to the single-core case, where most savings come from LLC Resizing, in a multi-core LLC static

power is not as dominant, hence we expect DVFS to play the most important role in energy savings.

Mix1 represents the pessimistic scenario that all applications are both cache and frequency sensitive. In this case, there is no potential for energy savings through DVFS and/or LLC Resizing, hence all policies choose to run on high frequency and LLC size, resulting in minimal EDP savings. *Mix2* demonstrates the case that applications are sensitive to frequency but insensitive to LLC size. In such a case, savings can be obtained only through LLC resizing, hence *DVFS only* policy does not yield any benefit. However, when all cores are active, LLC static power represents only 10% of total power (Figure 9b), hence once again potential for energy savings is limited even for the policies involving LLC Resizing. The reverse scenario (all applications are cache sensitive and frequency insensitive) is depicted in *mix3*, hence *DVFS only* achieves similar benefits to *DVFS & Resize* and *Resize only* fails to improve efficiency. *Mix4* involves cache-insensitive, frequency-insensitive applications that can take full advantage of both techniques. Interestingly, even the *Resize only* policy achieves better savings compared to *mix2* which also involves cache-insensitive applications. This is because applications of *mix4* are memory bound, therefore dynamic power is less dominant, increasing the impact of LLC static power into total power consumption. The rest of our workloads contain applications of mixed behavior in terms of cache and frequency sensitivity. *Mix7* shows why we need combined DVFS and LLC Resizing: *Resize only* policy is reluctant to reduce cache size, because LLC static power is not a big part of total chip power (Figure 9b). When DVFS is also applied, however, dynamic power is reduced and LLC static power becomes more important, hence it is beneficial to reduce cache size. This way, the *DVFS & Resize* policy gives a further 5% of EDP savings compared to *DVFS only*.

Finally, Figure 12b shows the two variations of *DVFS & Resize* policy, one without performance constraint (patterned) and one with 15% performance constraint (solid). Our models are accurate enough to succesfully enforce the 15% performance cap in each of the applications in the mix. On the other hand, the unconstrained policy is more aggressive and achieves higher power and EDP savings (e.g., 45% against 22% of EDP reduction for *mix3*).

6. Related Work

Karkhanis et al.[11] provide a fundamental insight into the importance of the parallelism of LLC misses for the performance of out-of-order execution. Eyerman et al.[5] exploit MLP to optimize the fetch policies for SMT processors. Qureshi et al.[23] exploit MLP to improve the replacement decisions in LLCs. Our approach relies on MLP to reduce static and dynamic power of a processor in two dimensions (scaling core V/f and LLC size in a synergistic manner).

The potential of DVFS in energy savings has been studied in a variety of research communities (from circuit to system designers). The most promising approach for DVFS management is the leading loads model, proposed by three independent groups[6, 13, 26]. This model predicts application performance under DVFS with an order of magnitude higher accuracy than previous models. Miftakhutdinov et al. [19] proposed dedicated hardware performance counters with the explicit goal of aiding the leading loads model, while [27] and [29] show how to approximate the leading loads model on commodity processors. Su et al.[30] proposed a linear regression model that takes into account power consumed by North Bridge and the temperature-dependent core idle power. Our DVFS mechanism is based on the leading loads model, however we argue that DVFS and LLC resizing must be applied in tandem. This stems from our observation that both DVFS and LLC Resizing must be driven by the MLP of LLC misses.

Cache Resizing is also a widely studied area. Yang et al.[35] proposed cache-set resizing, while hybrid (in both sets and ways) cache resizing was introduced in [36]. Keramidas et al. [12] proposed a methodology to eliminate the effect of transition misses, called hiccup misses (misses introduced due to resizing and not as part the normal cache operation). These approaches use the number of misses as performance indicator and they are not applicable in LLCs, where MLP is the dominant performance factor. [37] and [4] propose dynamic resizing of L1 caches using an extra set of cache tags (called miss and shadow tags respectively). These tags track the miss behavior in a different (target) cache configuration than the one that the cache currently operates. On the contrary, we focus on LLCs, we integrate the miss tags into the regular tags and our model offers a significantly higher coverage in the target cache configurations (not just one target configuration).

An orthogonal approach to cache resizing is cache partitioning among various concurrently executing applications in a multicore. Tam et al. [32] use hardware mechanisms for address sampling and post-processing software to compute reuse distance distributions and accordingly partition the LLC. [24] and [31] proposed cache partitioning mechanisms using counters associated with cache sets to help in tracking reuse distances. Moreto et al. [20] proposed a partitioning scheme based on MLP-cost. The latter approach requires a complete redesign of MSHRs and a new set of MSHRs in the front-end of the LLC. Our models can also be used for cache partitioning, by estimating performance for different partitioning decisions similarly to estimating performance for different cache sizes.

Finally, to the best of our knowledge, coordinate DVFS and cache resizing has been proposed in [16] and [18]. Both methods rely on empirical models (trained by costly profiling runs) and trial-and-error feedback mechanisms. Moreover, separate models and controllers are required for DVFS and resizing. In contrast, our proposal is based on analytical models and requires one controller for both techniques. More importantly, our models are flexible enough to effortlessly form a variety of different policies, whereas empirical methods are fine-tuned for a given optimization goal.

7. Conclusions

In this paper we present a unified analytical model for DVFS and Cache Resizing. We propose a methodology to track caches misses in different configurations without neglecting the hardware prefetcher. Moreover, we estimate MLP in any cache configuration and, based on this, we form a highly-accurate analytical model. A deeper examination of our model reveals that it can be easily augmented with a previously proposed DVFS model, hence we deliver a model that, from a single cache-frequency configuration can estimate performance for any configuration of interest.

To showcase the power of our model, we develop a framework that adapts cache size and core frequency to improve energy efficiency. We show that a co-ordinated management is crucial for maximizing efficiency, and we demonstrate the flexibility of our models by applying different variations of energy-efficiency policies, controlled by the OS or the application itself. Finally, we investigate the multi-core case and apply our method in a multi-workload environment.

References

[1] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.

[3] Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 493–504, Piscataway, NJ, USA, 2014. IEEE Press.

[4] K. Dayalan, M. Ozsoy, and D. Ponomarev. Dynamic associative caches: Reducing dynamic energy of first level caches. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 118–124, Oct 2014.

[5] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 240–249, Feb 2007.

[6] S. Eyerman and L. Eeckhout. A counter architecture for online dvfs profitability estimation. *Computers, IEEE Transactions on*, 2010.

[7] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 2009.

[8] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[9] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Abstracts*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.

[10] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[11] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[12] G. Keramidas, C. Datsios, and S. Kaxiras. A framework for efficient cache resizing. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 76–85, July 2012.

[13] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval based models for run-time dvfs orchestration in superscalar processors. In *Int. Conf. on Computing Frontiers*, 2010.

[14] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 253–262, New York, NY, USA, 2013. ACM.

[15] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2009.

[16] Kai Ma, Xiaorui Wang, and Yefu Wang. Dppc: Dynamic power partitioning and control for improved chip multiprocessor performance. *Computers, IEEE Transactions on*, 63(7):1736–1750, July 2014.

[17] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[18] Ke Meng, Russ Joseph, Robert P. Dick, and Li Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 177–186, New York, NY, USA, 2008. ACM.

[19] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *High Performance Embedded Architectures and Compilers*, pages 337–352. Springer, 2008.

[21] T. Nowatzki, J. Menon, C. Ho, and K. Sankaralingam. Architectural simulators considered harmful. *Micro, IEEE*, PP(99):1–1, 2015.

[22] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2001.

[23] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[25] B. Rountree. Theory and practice of dynamic voltage/frequency scaling in the high-performance computing environment. *Ph.D. dissertation, University of Arizona*, 2010.

[26] B. Rountree, D.K. Lowenthal, M. Schulz, and B.R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8, July 2011.

[27] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Int. Green Computing Conference*, 2011.

[28] Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 241–250. IEEE, 2012.

[29] Bo Su, Joseph L Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *Proc. USENIX Annual Technical Conf.(USENIX ATC)*, 2014.

[30] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L Greathouse, and Zhiying Wang. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 445–457. IEEE Computer Society, 2014.

[31] GE Suh, L Rudolph, and S Devadas. Dynamic cache partitioning for cmp/smt systems. *Journal of Supercomputing*, pages 7–26, 2004.

[32] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *Int. Conf. on Architectural support for Programming Languages and Operating Systems*, 2009.

[33] Sam Van den Steen, Sander De Pestel, Moncef Mechri, Stijn Eyerman, Trevor Carlson, L Eeckhout, Erik Hagersten, and D Black-Schaffer. Micro-architecture independent analytical processor performance and power modeling. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[34] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 577–589. IEEE, 2015.

[35] S.-H. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 147–157, 2001.

[36] S.-H. Yang, M.D. Powell, B. Falsafi, and T.N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 151–161, Feb 2002.

[37] Michael Zhang and Krste Asanović. Fine-grain cam-tag cache resizing using miss tags. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, ISLPED '02, pages 130–135, New York, NY, USA, 2002. ACM.