Energy reduction potential of a phase-based cache resizing scheme for embedded systems

Gilles Pokam, François Bodin

▶ **To cite this version:**

Gilles Pokam, François Bodin. Energy reduction potential of a phase-based cache resizing scheme for embedded systems. [Research Report] RR-5036, INRIA. 2003. inria-00071548

HAL Id: inria-00071548
https://inria.hal.science/inria-00071548

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Energy reduction potential of a phase-based cache resizing scheme for embedded systems

Gilles Pokam — François Bodin

N° 5036

Decembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5036--FR+ENG

Energy reduction potential of a phase-based cache resizing scheme for embedded systems

Gilles Pokam, François Bodin

Thème 1 — Réseaux et systèmes
Projets CAPS

Rapport de recherche n° 5036 — Decembre 2003 — 24 pages

Abstract: Managing the energy-performance tradeoff has become a major challenge on embedded systems. The cache hierarchy is a typical example of such a design target where this tradeoff plays a central role. With the increasing level of integration density, a cache can feature several billions of transistors, consuming a large proportion of the energy. In the same time however, it also allows to considerably improve the performance. Configurable caches are becoming the de-facto solution to deal efficiently with that problem. Such caches are equipped with artifacts that enable one to resize it dynamically. In the context of embedded systems, however, many of these artifacts restrict the configurability at the application level. We propose in this paper to modify the structure of a configurable cache to offer embedded compilers the opportunity to reconfigure it according to a program dynamic phase, rather than on a per-application basis. We show in our experimental results that the proposed scheme has a potential for improving the compiler effectiveness to reduce the energy consumption, while not excessively degrading the performance.

Key-words: cache, configurable architecture, low power, embedded systems

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Potentiel de réduction d'énergie d'une architecture de cache reconfigurable pour les systèmes embarqués

Résumé : La gestion de la consommation d'énergie devient un problème majeur dans les systèmes embarqués. A titre d'exemple, les caches dans les systèmes embarqués permettent d'améliorer les performances en maintenant une fraction importante du code source et des données sur la puce, réduisant ainsi la consommation d'énergie liée aux traffics mémoires. En même temps aussi, à cause du haut niveau d'intégration, un cache peut occuper jusqu'à 50% de la surface d'une puce, dissipant ainsi une large part d'énergie. De plus en plus des caches configurables sont donc introduits pour mieux pallier à ce problème. L'intérêt des caches configurables pour les systèmes embarqués est cependant encore assez limité. En effet, la plupart des solutions proposées actuellement n'offrent la possibilité de configurer un cache qu'une fois avant l'exécution de l'application. Nous proposons dans cet article de modifier la structure d'un cache configurable afin de permettre au compilateur de le reconfigurer en fonction des changements de phase dynamique de l'application. Nous montrons que le modèle de cache que nous introduisons offre un fort potentiel de réduction de la consommation d'énergie, tout en se gardant également de trop dégrader les performances.

Mots-clés : cache, architecture configurable, consommation d'énergie, systèmes embarqués

1 Introduction

As the demand for high-performance embedded systems increases, the challenge of managing power consumption in current embedded applications becomes a major concern. The cache hierarchy is a typical example of such a power/performance tradeoff design point. On the one hand, a larger cache allows to maintain an important fraction of the embedded code and the data workload on-chip, thus reducing the amount of memory traffic and thereby improving the performance and the power consumption. On the other hand, however, on some embedded processors, the cache memory accounts for up to 80% of the total transistor count and for about 50% of the total chip area [1], making the cache memory subsystem an important source of power dissipation.

Recent researches on this area have focused on the design of configurable caches [4, 5, 6, 7, 9, 10]. The main motivation behind a configurable cache is to allow one to adapt the cache size requirement of a running program to a desired power/performance tradeoff. However, former configurable cache proposals for embedded systems [4, 5, 7] have only considered configuration on a per-application basis. The fundamental drawback with this approach is that, an optimal cache size, independently of its performance over an application, does not exist, whereas each application simply exhibits varying dynamic cache behaviors [2, 3]. In the context of VLIW-based embedded systems, another important aspect with regard to this is given by the design complexity of the proposed configuration scheme, since the compiler plays a central role in obtaining good performance. In particular, the concern here is determining the tradeoff design point where to balance between the architecture and the compiler to achieve a fair configuration flexibility.

This paper is a first effort towards the resolution of the problems exposed above. First, a model of a hybrid configurable cache design is proposed as a shortcut to two current proposals, to allow a cache to be reconfigured on a per-phase basis rather than at the application level, with only minor hardware modifications, keeping the design complexity simple. Second, based on the proposed model, the potential benefits of a fine-grain cache size adaptation scheme is explored that can be used at the compiler level for automatically characterizing the different cache size requirements of a program phase. Our model considers a great degree of flexibility, providing the compiler with the opportunity of resizing a cache along its size and/or degree of associativity. In addition, the model can follow program's dynamic behavior since it is well known that programs usually execute as a series of phases [2].

The remainder of this paper is organized as follows. Section 2 provides a review of configurable cache designs. In Section 3, we detail our model of a hybrid recon-

figurable cache architecture. The compilation support to our hybrid cache model is presented in Section 4. We present our experimental results in Section 5. Related works are discussed in Section 6, and Section 7 concludes.

2 Configurable cache designs

Configurable caches offer a powerful alternative for reducing the energy dissipation of conventional caches. The basic idea is to allow a cache memory system to be adapted to the cache size requirement of a running program. The various proposals of configurable cache architecture principally differ in their resizing granularity and hardware complexity.

In [6], Albonesi proposes to naturally partition a set-associative cache along its tag and data ways. Energy saving is achieved by allowing cache ways to be disabled/enabled on the need, according to the cache size requirement of the application. The hardware needed to implement selective-ways is minimal, only a software register mask to enable/disable cache ways accompanied with the corresponding control logic are required. The method is however only practical for set-associative caches. The configurable cache design proposed in [8] is somewhat similar to selective-ways in that the cache partitioning scheme used is the same. However, instead of disabling the unused cache sections, the authors suggest to transfer useful tasks to them, e.g. instruction reuse for media processing.

Another approach of a configurable cache scheme consists in partitioning a cache along its sets. This technique was originally proposed in [10]. Compared to selective-ways, this approach presents the advantage to be useful also in direct-mapped caches, which is not the case for the former. However, the proposed implementation is complex and the required hardware very expensive. To accommodate to the smallest and/or largest addressable number of sets to which a cache can be resized, the required number of tag bits can often exceed the one of a conventional cache of equal size. For instance, in a 32K 4-way cache with 8 tag index bits, resizing the number of sets from 256 to 1024 requires maintaining a cache implementation with 10 index bits. This not only complicates the cache lookup logic, but also adds to the chip area cost.

A more recent work by Zhang et al. [7] proposes to exploit the way partitioning scheme of a set-associative cache to reconfigure it as either a direct-mapped cache or a set-associative cache of lower associativity degree. The proposed configuration scheme exploits a technique called way-concatenation which allows cache ways to be merged, while still retaining the full cache capacity but with reduced set-associativity.

size/#sets	1024	512	256
32K	DM	2-way	**4-way**
16K		DM	2-way
8K			DM

Table 1: Cache size granularities for a 4-way 32K cache with each bank having 8K.

This approach permits to reduce dynamic energy since at same cache size, lower associativity caches have fewer switching activities than higher associativity caches. Moreover, the hardware needed to implement the method is minimal.

3 Cache model description

We describe in this section the architectural model of the reconfigurable cache assumed for this study. While this model can apply to both data caches and instruction caches, only the data cache is considered. In addition, we also assume the data cache implements a LRU replacement policy and a write-through strategy with non-write allocate.

3.1 Motivation

The main motivation behind the proposed model is to emphasize the most important application-specific cache architectural tradeoffs involved during program execution. To do this, we consider a cache with a fixed line size and we observe that the performance of such a cache is mainly dictated by its size and degree of associativity. From a software perspective, we would then like to select for each application, the cache with the lowest degree of associativity and the smallest size that reduces the access frequency to the upper level caches. However, since programs have varying dynamic cache behaviors, they must also feature varying dynamic cache size and/or degree of associativity. Thus, instead of selecting these parameters on a per-application basis, we would then prefer to tune them according to a program dynamic phase.

The idea is to use a combination of schemes that allows a cache to be reconfigured along its size and associativity to provide both in one. We do this by exploiting the variability in the cache size and degree of associativity provided by combining the selective-way scheme [6] and the way-concatenation technique [7]. Such an hybrid scheme can provide fine-grain cache sizes at various degree of associativity. Table 1 shows a subset of possible cache configurations that can be exposed to the com-

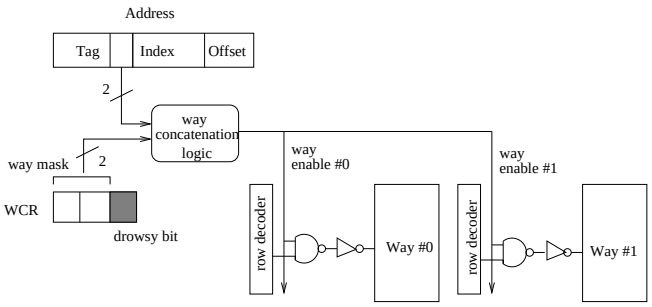

Figure 1: Baseline architecture of a 2-way associative cache.

piler. Starting from a baseline cache configuration of 32K 4-way set-associative, we can feature anyone of the highlighted configurations by either activating the way-concatenation scheme (following the horizontal lines), the selective-way scheme (following the vertical lines) or both. For example, we move to the 16K direct-mapped configuration by either concatenating 2 banks (32K 2-way) and then selecting only one of the two, or selecting two (16K 2-way) active banks and then concatenating them.

3.2 Baseline model

Our model builds upon the way-concatenation scheme introduced in [7] and extends it to include a flexible selective-way scheme which allows the size of a cache to be adjusted at runtime. Basically, the way-concatenation scheme allows one to select the number of cache ways m that must be activated on each cache lookup, where each selected way is virtually a multiple of the size of a cache way in the n-way case, n being the number of available cache ways. For instance, if one way is active, it has virtually four times the size of the 4-way case. A configuration register is provided in which the number of active ways m is set. A way concatenation logic is then in charge of carrying the active/inactive way-enable signal to each one of the n cache ways. In Figure 1, the two high-order bits of the way concatenation register (WCR) represent the configuration register in the baseline architecture. Figure 1 also shows the way-enable signals as driven by the way concatenation logic.

3.3 Architectural modifications

Associativity dimension. The main issue of concern we address at this stage is preserving the cache coherency across different cache configurations, while minimiz-

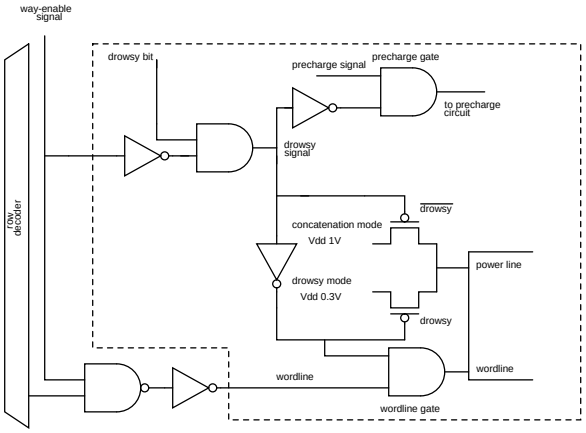

Figure 2: Drowsy cache line circuit.

ing the reconfiguration time. Consider, for instance, the reconfiguration scenario illustrated in Figure 3. In phase i, corresponding to a 2-way cache configuration, the way-concatenation logic activates bank 0 and bank 2 when @A is referenced. In this case, @A hits in bank 0. In phase $i+1$, however, the cache configuration changes to a direct-mapped cache and @A is write accessed in bank 1. At this stage, there are two possible locations for @A, the old one in bank 0 and the new one in bank 1.

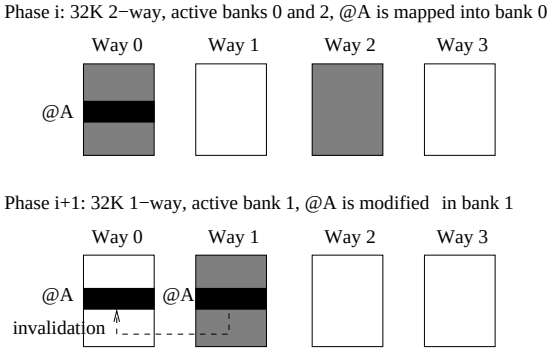

Figure 3: Reconfiguration scenario.

A possible way to overcome the cache coherency problem illustrated above is to make the tag and status arrays always accessible. This implies that only the data array can be activated/deactivated by the way-concatenation logic. The tag and

status arrays therefore still continue to behave like in a conventional cache. The actions of the cache controller can then be modified to access all tag arrays on each write request to set the corresponding status bit to invalid whenever the referenced address hits in one of the bank. This scenario is illustrated in Figure 3 with the dotted arrow line indicating the action of the invalidation signal in the tag array. Future accesses to the invalidated data will cause the new data to be provided by the upper level cache hierarchy. A write-through cache is assumed to guarantee that the provided data is the most recent one. This implementation can be done via a special instruction in software to force this behavior or it can also be done transparently in hardware.

Cache size dimension. We augmented the way-concatenation architecture to include a drowsy bit, represented by the low-order bit of WCR shown in Figure 1. The drowsy bit is intended to control the activation/deactivation of the selective-way scheme (drowsy mode). This mode assumes that the machine supply voltage can be dynamically scaled to higher values of the threshold voltage V_T. As in the gated-Vdd scheme [13], this mode allows to reduce the leakage energy due to the scaling of the supply voltage which in turn also causes the leakage current to be reduced as a by-side of the short-channel effects. In contrast to the gated-Vdd scheme, however, scaling the supply voltage permits to maintain the state of the memory cell. Therefore, when reducing the cache size, we do not need to completely disconnect a cache bank which may otherwise cause the loss of the data stored into it. Note that, the drowsy mode only applies to the data array, as explained above. This solution has been preferred to avoid the one cycle wake-up delay to bring a tag way out of drowsy mode on each write access. The fact of permanently maintaining the tag array in a non-drowsy mode has a negligible impact on the leakage energy since the tag ramcells count for less than 4.2% of the total area in a 32KB 4-way cache with 32B line size.

Figure 2 reflects the changes introduced into the cache line to accommodate to the drowsy mode. The drowsy bit is ANDed with the way-enable signal of each cache line. An entire cache way may then be put into drowsy mode depending on the status of the way-enable signal and the drowsy bit. In particular, this happens when the drowsy bit is set and the corresponding way-enable signal is unset. In such case, the supply voltage to each cache line switches to the lower voltage, putting the entire bank into drowsy mode. In the other cases, the supply voltage to each cache line is set to the normal voltage, bringing the entire cache bank out of drowsy mode.

way-mask value	drowsy bit state	cache config.
0	0/1	32K1W/8K1W
1	0/1	32K2W/16K1W
2	0/1	32K2W/16K2W
3	0	32K4W

Table 2: Effects of the **WCRMOV** instruction.

3.4 Design cost

To drive the drowsy signal to each cache line, we added an inverter and a AND gate. By assuming a memory cell dimension of 1.84 x 3.66 um, this results to approximatively 2 memory cells per cache line. According to [14], the voltage controller adds about 3.35 memory cells, assuming a memory cell layout of 6.18 x 3.66 um. The two inverters, one in the voltage controller and the other in the precharge circuit, add an equivalent of 1 more memory cell per cache line. Finally, the wordline gating circuit accounts for 1.5 additional memory cells, making a total of 7.85 memory cells overhead per cache line. Overall, for a cache size of 32K and a line size of 32B, this makes an area overhead of less than 3%. Note that, in comparison to the circuit shown in [14], there is no need to use a drowsy bit on each cache line since the drowsy signal is directly derived from the way-enable signal which is driven to each cache way. Using a drowsy cache adds however some performance penalty. When a drowsy cache way is activated, the voltage controller to each cache line simultaneously retires from the low voltage to set the memory cell power line to the normal voltage. This takes one additional clock cycle.

3.5 ISA support

The ISA support to the model presented above can be resumed to a simple WCR modify instruction, denoted by **MOVWCR**, to read/write the content of WCR shown in Figure 1. Given that such an instruction is provided by the ISA, Table 2 illustrates how this instruction can be used to feature the different cache configurations shown in Table 1.

4 Compilation support

We describe in this section the compilation support that makes use of the model presented previously. In particular, we show how the compiler may characterize the cache size requirement of a dynamic program phase.

4.1 Program cache size requirements

The characterization of the cache size requirement of a dynamic program phase is performed on a trace of address references previously extracted by means of profiling. Since an embedded system is often designed to run a few types of applications, it is worth to spend a fraction of time to optimize each embedded application intensively. In such case, the time required to profile and pre-process the embedded applications can be sustained.

Our approach to program profiling and trace processing consists in collecting the dynamic LRU-stack profiles $P_{\Pi_i}(map_j(x))$ and $E_{\Pi_i}(map_j(x))$, explained later, of the running program, at each fixed sample interval Π_i. The variable x in the previous expressions corresponds to the LRU-stack depth. By exploiting the cache inclusion property, assigning different values to the variable x allows to simultaneously evaluate alternative cache memory configurations that share the same set-mapping function map_j. Thus, by also varying the set-mapping function map_j, we can increase the range of alternative cache configurations that can be simultaneously evaluated in a one pass simulation through the address trace [15]. We assume for the rest of this study that the caches we model do no prefetching, have the same block size and use the LRU replacement policy.

4.1.1 Cache size performance profile, $P_{\Pi}(map(x))$

At each sample interval Π_i, $P_{\Pi_i}(map_j(x))$ defines the performance profile of a cache with set-mapping function map_j and LRU-stack distance x. This performance profile can be seen as the number of dynamic references that hit in all cache configurations with the same set-mapping function map_j and with the LRU-stack distance $d \leq x$.

4.1.2 Cache size energy profile, $E_{\Pi}(map(x))$

Similarly, the expression $E_{\Pi_i}(map_j(x))$ defines the dynamic energy profile of a cache with set-mapping function map_j and LRU-stack distance x, at each sample interval Π_i. We define the dynamic energy per sample interval as follows:

$$E_{\Pi_i}(map_j(x)) \quad = \quad P_{\Pi_i}(map_j(x)) * E_c \tag{1}$$

$$+ \quad |\Pi_i| * E_T + N_{\Pi_i} * E_d$$
$$+ \quad (|\Pi_i| - P_{\Pi_i}(map_j(x))) * (E_c + E_m)$$

where

$$E_m = dcache2mem\_energy\_ratio * E_c \tag{2}$$

In the equation (1), E_c is the dynamic energy on each cache access, E_m the dynamic energy per memory access, E_d the dynamic energy per drowsy transition, E_T the dynamic energy per each tag access, and N_{Π_i} the number of transitions to/from drowsy mode within the sample interval Π_i. N_{Π_i} is measured by means of monitoring all bank transitions within two consecutive dynamic cache memory accesses, reporting only those bank transitions whose prior state was set to drowsy. The first expression in (1) models the dynamic energy due to a hit in the cache. The second and third expressions respectively models the energy due to accessing the tag part and the energy due to the drowsy mode transitions. Finally, the last expression models the energy due to accessing the memory on a miss event and on each write-through access to the cache. In (2), we estimated the $dcache2mem\_energy\_ratio$ constant to be 50 for current cache generation.

4.2 Management of reconfigurability

The reconfiguration can be undertaken by the compiler in the following manner. Assuming the ISA support introduced in Section 3.5 is given and that the application working set has been partitioned into different cache configurations, the compiler may insert reconfiguration instructions in the code at each position where a new cache configuration begins, with the appropriate parameters conveniently set (e.g drowsy bit state, way-mask). This step can be done simultaneously to the cache size calibration of the application's working set described in Section 5.3 or within a separate compilation pass. Since in this paper we focus on the potential benefit provided by such a scheme, we do not address the compilation issues of automatically inserting the reconfiguration instructions in the code. This research will be delegated to future works.

5 Experimental setup and results

This section presents a preliminary evaluation of the cache size adaptation scheme introduced in Section 3 and Section 4. Our simulation platform and the benchmarks

Parameter	Value
Issue width	4
Integer ALU	4
Multiplication units	2
Load/Store unit	1
Branch unit	1
data cache	32K 4-way
data cache line size	32B
data cache access latency	1 cycle
data cache replacement policy	LRU
memory access latency	20 cycles

Benchmark	Suite	Datasets
fft	MiBench	large
gsm	MiBench	large
susan	MiBench	large
mpeg	mediabench	test
epic	mediabench	test_image
summin	Powerstone	custom
whestone	Powerstone	custom
v42bis	Powerstone	custom

Figure 4: (a) Lx microarchitecture parameters; (b) Benchmarks used and number of accesses to data cache (in million).

used are first described. Then, an analysis of the performance and energy profiles is introduced that serves as a prelude to the application's working set partitioning algorithm presented hereafter. Preliminary results are commented at the end.

5.1 Simulation platform and benchmarks

Our simulations were carried out on the Lx platform [18]. The Lx platform belongs to a family of customizable multi-cluster VLIW architectures. The implementation used in this study features a 4-issue width processor. The details of the processor microarchitecture parameters are shown in Figure 4-a. We evaluated our cache size adaptation scheme with different applications collected from MiBench [17], mediabench [21] and Powerstone [20] suites. All the chosen applications were compiled with the Lx native compiler, with the optimization level 3, and then run until completion. Figure 4-b shows an overview of each benchmark together with the datasets used. Some of the benchmarks from these suites that we did not considered were not able to be compiled with the Lx native compiler or were exhibiting close behaviors to some applications that we already selected. For the rest of this study, we focus our analysis on the impact of the proposed scheme on the data cache.

Our simulation parameters were obtained by means of CACTI [16] and Hotleakage [19]. In particular, we extended CACTI to include the leakage energy functions of the Hotleakage tool. We then employed the resulted modified CACTI tool to estimate the dynamic energy per cache access for each simulated cache configuration,

Parameter	Value
process technology	0.07 um
normal supply voltage	0.9 V
drowsy supply voltage	0.3 V
processor clock speed	5.6 GHz
drowsy transition latency	1 cycle
32k 4-way dynamic energy/access	0.294 nJ
32k 2-way dynamic energy/access	0.173 nJ
32k 1-way dynamic energy/access	0.110 nJ
16k 1-way dynamic energy/access	0.104 nJ
16k 2-way dynamic energy/access	0.164 nJ
8k 1-way dynamic energy/access	0.104 nJ
drowsy energy/transition	0.256 pJ
gated-Vdd leakage energy/cell	0.245 fJ
drowsy leakage energy/cell	0.308 pJ
normal leakage energy/cell	0.835 pJ

Table 3: Simulation parameters.

as well as the leakage energy per cell for each simulated leakage energy reduction technique. The dynamic drowsy transition energy was derived based on the results published in [14]. Table 3 gives an overview of the full simulation parameters that apply to this study.

5.2 Analysis of the energy and performance profiles

During program execution, we recorded at each sample interval Π_i, where $\Pi_i = 100K$ cycles $\forall i$, the energy and performance values for each modeled cache configuration, as described in Section 4.1. In order to emphasize the different energy/performance tradeoffs between the modeled cache configurations, we graphed in Figure 5 and Figure 6 the cumulated energy values versus the number of cumulated cache misses encountered in each sample interval of the program run. The x-axis records the logarithmic scale of the cumulated number of cache misses obtained in each sample interval (e.g. a 3 in the x-axis means 10^3 misses). Each point of the x-axis is associated a value in the y-axis corresponding to the cumulated amount of dynamic energy consumed (also in logarithmic scale) up to that sample interval. Let us for

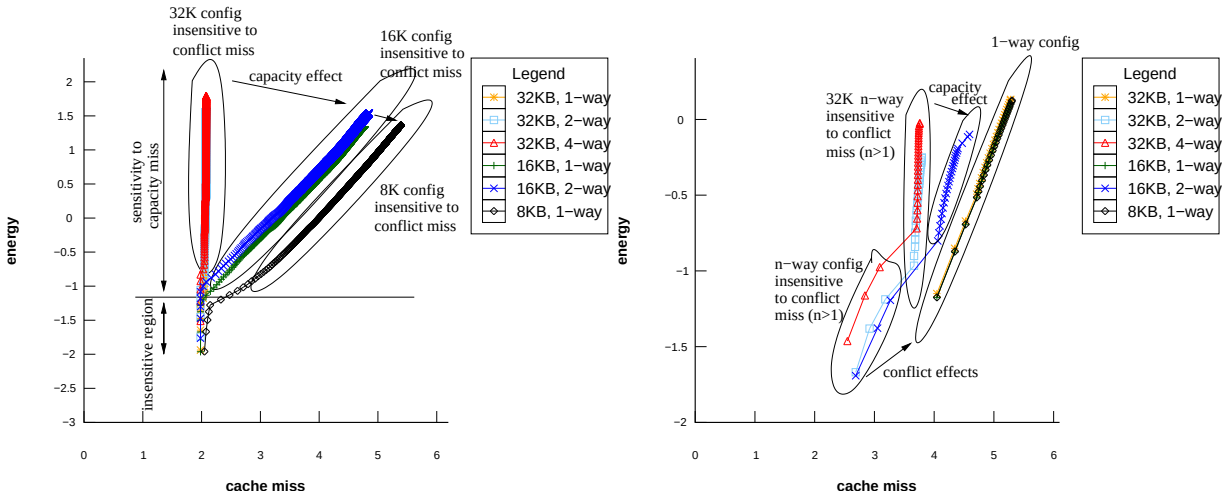

Figure 5: (a) gsm energy/performance profile; (b) fft energy/performance profile.

instance consider the *gsm* and *fft* applications shown in Figure 5-a and Figure 5-b, respectively.

In Figure 5-a, we can observe that there exists a threshold at which the different modeled cache configurations are clustered according to their size, independently of the degree of associativity. In particular, we can distinguish three clusters: one with all 32K configurations, another with all 16K configurations, and the last with the 8K configuration. Configurations that belong to the same cluster are insensitive to the degree of associativity. Within each cluster, the distinguishing factor is indubitably the dynamic energy consumption due to the cache hits since the number of cache misses are relatively the same. In this latter case, the amount of dissipated energy is tight to the architecture of the featured cache configuration and is mostly a function of the cache size. A desired energy/performance tradeoff can then be achieved by moving from one cluster to the other, as indicated by the arrow shown in that figure. This comes at the cost of some performance degradation.

In Figure 5-b, we can also observe that two main cache clusters can be distinguished: one including all n-way cache configurations with $n > 1$ and the other including all direct mapped cache configurations. These two clusters differ essentially in the degree of associativity. As the program execution proceeds, the increasing effect of the capacity misses forces the first cluster to be further splitted into two distinct clusters: one which includes the n-way 32K cache configurations with $n > 1$

and the other which is constituted of the 2-way 16K cache configuration. In this case, the clusters are splitted according to the cache capacity, independently of the degree of associativity. Again, in each cluster, the energy consumption due to the hits also serves as the main distinguishing factor.

We have observed that this property is rather common to most programs, as it can be seen in Figure 6. These examples prove that the dynamic working set of a program can be arranged so as to take benefit of the inherent working set sensitivity to the conflict or capacity miss, to save more energy. We show in the next section how we exploited this principle to devise an algorithm that tailors the cache memory accordingly.

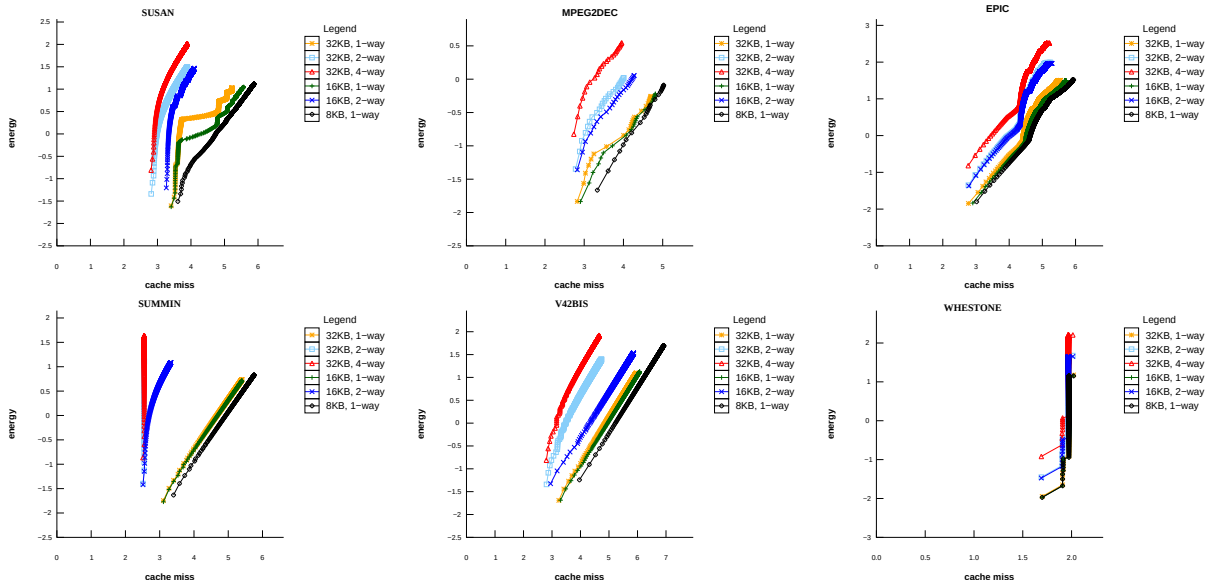

Figure 6: Energy and performance profiles

5.3 Working set partitioning algorithm

The intuition behind the working set partitioning algorithm follows the idea described in the previous section. The objective is to achieve a partitioning of the application's working set into clusters of cache configurations sharing the same sensitivity to either

the conflict or the capacity misses. This latter point is mainly dictated by the need of keeping the number of reconfiguration points smaller enough in order not to impact the performance and the energy. In the worst case, the cache may be reconfigured at the beginning of each sample interval, causing unnecessary reconfigurations if the selected configuration does not provide considerable benefits. In addition, the performance and the energy consumption may be also impaired by the excessive number of inserted reconfiguration instructions - which may grow the code size and raise the energy consumption - and the additional number of cache misses induced by changing cache configurations.

The partitioning is done relative to the base cache configuration. In particular, a cluster of cache configurations with identical sensitiveness to the conflict/capacity miss is constructed from two cache configuration points B and C if each one of them belongs to the closest vicinity of the other, with respect to a reference point A of the base cache configuration.

Let us consider C_{base} and C_k as being the set of values collected at each sample interval Π_i for the base cache configuration C_{base} and for each simulated cache configuration C_k, respectively. The expressions of C_{base} and C_k are defined as follows:

$$C_k = \{(P_i^k, E_i^k), 0 < i \leq N\} \tag{3}$$

$$C_{base} = \{(Pbase_i, Ebase_i), 0 < i \leq N\} \tag{4}$$

In the above expressions, N represents the number of sample intervals Π_i. In order to partition the working set into similar sensitive cache configurations, we define for each simulated cache configuration C_k, a Manhattan distance vector V_k, as follows:

$$V_k = (v_1^k, v_2^k, ..., v_N^k), \tag{5}$$

where

$$v_i^k = |P_i^k - Pbase_i| + |E_i^k - Ebase_i| \tag{6}$$

Two cache configuration points (P_i^{k1}, E_i^{k1}) and (P_i^{k2}, E_i^{k2}) belong to the same cluster if their respective Manhattan distance value is related by the following relation:

$$|v_i^{k2} - v_i^{k1}| < \tau \tag{7}$$

τ is a threshold value that can be adjusted to strengthen or relax the clustering criterion. Once the cache configuration points have been clustered into partitions

of equal sensitiveness, each partition is chosen a representative cache configuration based on the best performance to energy ratio of each cluster of cache configurations.

The performance to energy ratio of each cluster is computed based on the value of the last sample point ($i = N$) in the considered cluster. The idea is to capture the relative amount of performance degradation corresponding to a given energy budget. For this, the ratio $\frac{P-Pbase}{Ebase-E}$ of each cache configuration belonging to a cluster is evaluated against each other. A smaller ratio is preferred since this would imply that for a given power budget, the performance is better. Then, for two clusters that span the same working set size, we choose the cache configuration of the representative partition element which has the best performance to energy ratio. The ISA instruction introduced in Section 3.5 can then be inserted at the appropriate working set frontier to enable the corresponding cache configuration.

5.4 Preliminary results

This section presents the results obtained by evaluating the proposed cache resizing scheme. The evaluation discussed in the remainder of this section is centered around three different performance aspects: the dynamic energy reduction, the leakage energy reduction and the performance degradation estimated in terms of increased total cycle counts.

Dynamic energy reduction. The leftmost side of Figure 7 shows the dynamic energy consumption results for the proposed cache resizing scheme. For the purpose of comparison, we also evaluated the dynamic energy consumption of the best performing cache configuration. It can be seen from the figure that the proposed hybrid scheme can indeed reduce the energy consumption in some cases. Looking precisely at them, e.g *gsm*, *susan*, *summin*, *mpeeg*, *epic* and *v42bis*, we observe that they correspond, to some extent, to the cases where there exists a source of working set size variation in the program execution. This mainly explains why the working set can be ideally partitioned into clusters of different cache configurations. In these examples, the energy consumption can be further reduced from 5 up to 12% compared to the best performing cache configuration. However, in the other cases where the working set shows little or no variation, the proposed hybrid scheme provides no benefit. This is the case of *fft* and *whestone*.

Leakage energy reduction. We estimated the leakage energy, E_{Leak}, of a program as follows:

$$E_{Leak} = e\_leak_i * Ncell * T_{cyc}$$

In the above expression, e_leak_i represents the leakage energy per cell for each one of the simulated leakage reduction technique i, Ncell the number of cells in the cache and T_{cyc} is the total number of cycles to execute the given program. T_{cyc} is computed as the sum of the number of cycles required to execute the program without any data cache stalls, plus the estimated data cache miss times the miss penalty. Figure 8 illustrates the leakage energy of the cache configurations show in Figure 7, left. We calculate the leakage energy of the best performing cache configuration by employing a gated-Vdd-based technique. It can be observed from the figure that the proposed hybrid scheme can reduce the static energy by more than 80%. This is a substantial reduction since the leakage energy of future caches generation are predicted to consume as much as 50% of the total power consumption [19]. The advantages of the hybrid scheme are best highlighted on *fft*, *gsm*, *susan*, *summin*, *epic* and *v42bis*. The best cache configuration is however superior to our scheme whenever the capacity of the cache can be reduced over the entire program run. This is the case of *whestone*. In this latter example, the gated-Vdd scheme considerably reduces the static energy compared with the drowsy mode. However, because this case is not the common, we believe our proposed hybrid scheme offers a more flexible alternative for many other applications.

Performance degradation. We evaluated the performance degradation in terms of the number of additional clock cycles required to execute a program. The simulated results are shown in the rightmost side of Figure 7. The primary causes of performance degradation in the proposed scheme are due to the one cycle delay of the drowsy transitions and the cache misses induced by changing a configuration. Our results are relatively high in some cases because we actually have considered the worst case in which a drowsy transition may occur even within a single phase. This is indeed inherent to the architecture since two data addresses may eventually be mapped to different combinations of cache bank in the same phase, causing unnecessary drowsy transitions. This is mainly reflected in *susan* and *mpeg* where the degradation are the worst, 35% and 31% respectively. From within this additional number of cycles, more than 65%, in average, are due to the drowsy transitions, the remaining part being due to the additional number of cache misses. A more efficient solution will therefore consist in choosing the set of invariant cache banks that will remain active throughout a complete program phase. This solution provides the benefit of eliminating the superfluous drowsy transitions, but at the cost of increasing the number of cache misses due to the invalidated data that may eventually be accessed in other configurations.

6 Related work

Our work is primarily concerned with research related to cache size adaptivity. In this sense, the work in [9, 11, 12] bear some similarities with our own. These research share the particularity that some means of hardware adaptation scheme is required to allow a search of the optimal solution. In [9], the authors adapt the cache size of a L1/L2 or L2/L3 memory hierarchy in reaction to the sensitivity of a running program to some performance metrics collected dynamically, involving the IPC, the cache miss ratio and the branch frequency. The authors rely on the selective-way architecture [6] to accordingly enable/disable cache ways. This work is intended to general purpose systems featuring several levels of cache memory hierarchy. The proposed cache resizing algorithm can however as well be used in the context of embedded systems, provided some means of dynamic performance monitoring is available, e.g software accessible performance counters for each metric. In the same order, [12] reformulates the cache resizing adaptivity algorithm of [9] to use instead working set signatures to capture phase changes and estimate the size of a working set. This solution however also requires an extra hardware effort. Yang et al. [11] proposed a work similar to ours. They rely on the cache resizing schemes of [6] and [10] to propose a hybrid cache of superior resizing granularity than either one of them. The hardware implementation cost of the selective-set scheme is however very expensive to be integrated on a embedded system. In addition, the proposed hybrid cache has a more restrictive cache resizing granularity for direct-mapped cache configurations (8K in the paper for a cache size of 32K). In addition to this, we should notice that our work primarily addresses embedded systems. We seek therefore a low-cost, software-based cache resizing adaptivity scheme. Though our objectives are the same, the different application domains impose us to look for different solutions.

Zhang et al. [7] also presented how the way-concatenation scheme could be used together with a selective-way-based scheme to further reduce energy. The main difference between our approach and the one proposed by Zhang is essentially on the applicability of cache resizing. Zhang et al. look for the cache configuration that gives the best performance on a per-application basis, and therefore only configure the program once at startup time. Thus, the variations in cache size requirements within the running program is not taken into account. We seek instead to adapt the cache memory to meet the dynamic cache size requirements of the running program. This difference infers some divergences in the implementation decisions of the selective-way scheme. Zhang et al. propose to use a circuit technique called gated-Vdd [13] to implement their selective-way scheme. Gated-Vdd permits to reduce the cache leakage energy by gating off the supply voltage to the unused cache memory cells.

However, this technique does not preserve the memory cell state, causing the stored data to be loss. This, in addition to the coherency problem we addressed in Section 3.3, are serious hindrance to make their scheme reconfigurable on a per-phase basis.

7 Conclusions and future works

This research has been primarily motivated by the fact that future embedded systems workload will soon become more and more sophisticated [3], requiring even more aggressive, but at low cost, cache resizing architectures. To this purpose, we have proposed to modify the structure of a configurable cache to offer embedded compilers the possibility to reconfigure the underlying cache memory according to the cache size requirement of a dynamic program phase. We showed that with the proposed cache resizing scheme, some reduction in the dynamic and static energy can be realized. In essence, we proved that this energy reduction is significant for applications showing a dynamic working set size variation. In particular, we showed that in such cases, the application's working set can be classified according to a program property we called conflict/capacity miss insensitiveness. We presented simulation results that demonstrated this property is rather common to most programs. In the light of this model, we explored a compiler strategy that may take advantage of this property to partition the application's working set into clusters of similar cache sensitive configurations, to save more energy.

Though the encouraging results obtained in this study, the proposed cache resizing scheme still need to be investigated in more details. In particular, our future works include developing a compilation technique for automatic cache resizing and reducing the performance penalty due to the drowsy mode. For this purpose, we are experiencing with more aggressive cache bank access policy. We believe that reducing this performance penalty may make this scheme very attractive for embedded systems.

References

[1] John Hennessy. The Future of Systems Research. IEEE Computer, pages 27-33, August 1999.

[2] T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical Report CS99-630, University of California, San Diego, August 1999.

[3] N. Slingerland and A. J. Smith. Cache Performance for Multimedia Applications. In Proc. ACM Intl. Conf. on Supercomputing, pp. 204-217, Jun. 2001.

[4] Fumihiko Hayakawa, Hiroshi Okano and Atsuhiro Suga. An Eight-Way VLIW Embedded Multimedia Processor with Advanced Cache Mechanism. In Proceedings of the Third IEEE Asia-Pacific Conference on ASICs (AP-ASIC2002), pp. , Taipei, Taiwan, August 2002

[5] Malik, A., B. Moyer, D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In Int. Symposium on Low Power Electronics and Design, pp. 241-243, 2000.

[6] D.H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In Proceedings of the 32nd International Symposium on Microarchitecture, November 1999.

[7] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In 30th International Symposium on Computer Architecture, June 2003.

[8] N. P. J. Parthasarathy Ranganathan, Sarita Adve and Norman P. Jouppi. Reconfigurable Caches and Their Application to Media Processing. In Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, June 2000. IEEE Computer Society.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Processor Architectures. In Proceedings of MICRO-33, pages 245–257, Dec 2000.

[10] S-H. Yang, M. Powell, B. Falsafi, K. Roy and T. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High Performance I-caches. In Proc. of the Int. Sym. on High Performance Computer Architecture (HPCA), Jan. 2001.

[11] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, and T. N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In IEEE International Symposium on High Performance Computer Architecture (HPCA), February 2002.

[12] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, May 2002.

[13] M.D. Powell, S-H. Yang, B. Falsa, K. Roy, T.N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In ACM/IEEE International Symposium on Low Power Electronics and Design, 2000.

[14] Kristian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In 29th Annual International Symposium on Computer Architecture, May 2002.

[15] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. IEEE Transactions on Computers, 38(12):1612–1630, December 1989.

[16] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing Power, and Area Model", DEC Western research Lab Report 2001/2.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization, pp. 3-14, Dec. 2001.

[18] P. Faraboschi, G. Brown, Joseph A. Fisher, G. Desoli, and Fred Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In Proc. of the 27th Intl. Symposium on Computer Architecture, June 2000.

[19] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, Mar. 2003. 34.

[20] Jeff Scott, Lea Hwang Lee, John Arends and William Moyer. Designing the Low-Power M.CORE Architecture. In Power Driven Microarchitecture Workshop, Spain, June 28th 1998.

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In Proc. of the 30th Annual International Symposium on Microarchitecture, Dec 1997.

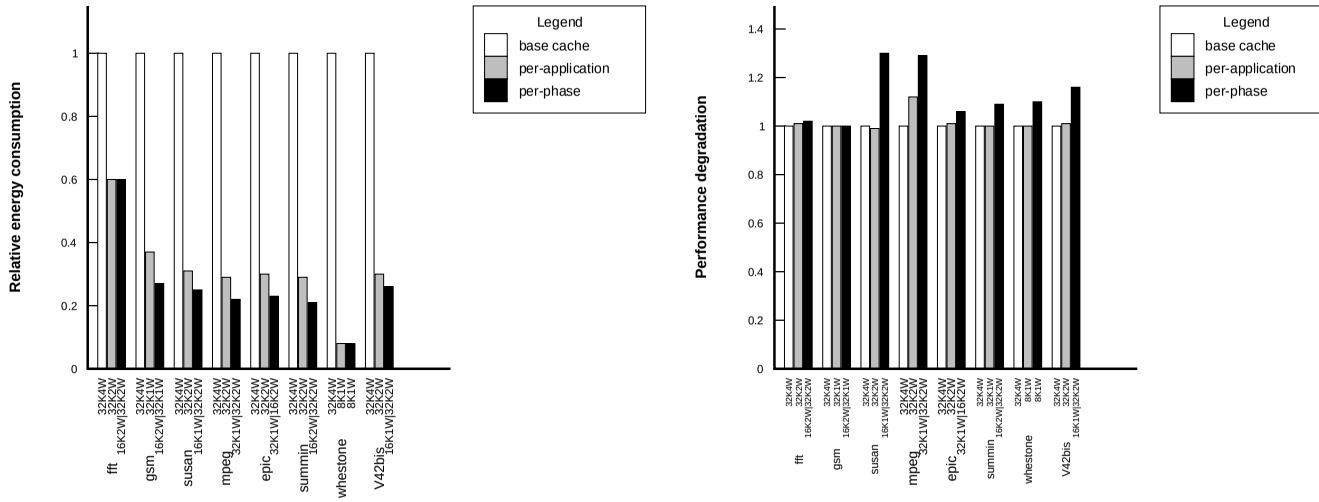

Figure 7: (left) Dynamic energy consumption; (right) Performance degradation.

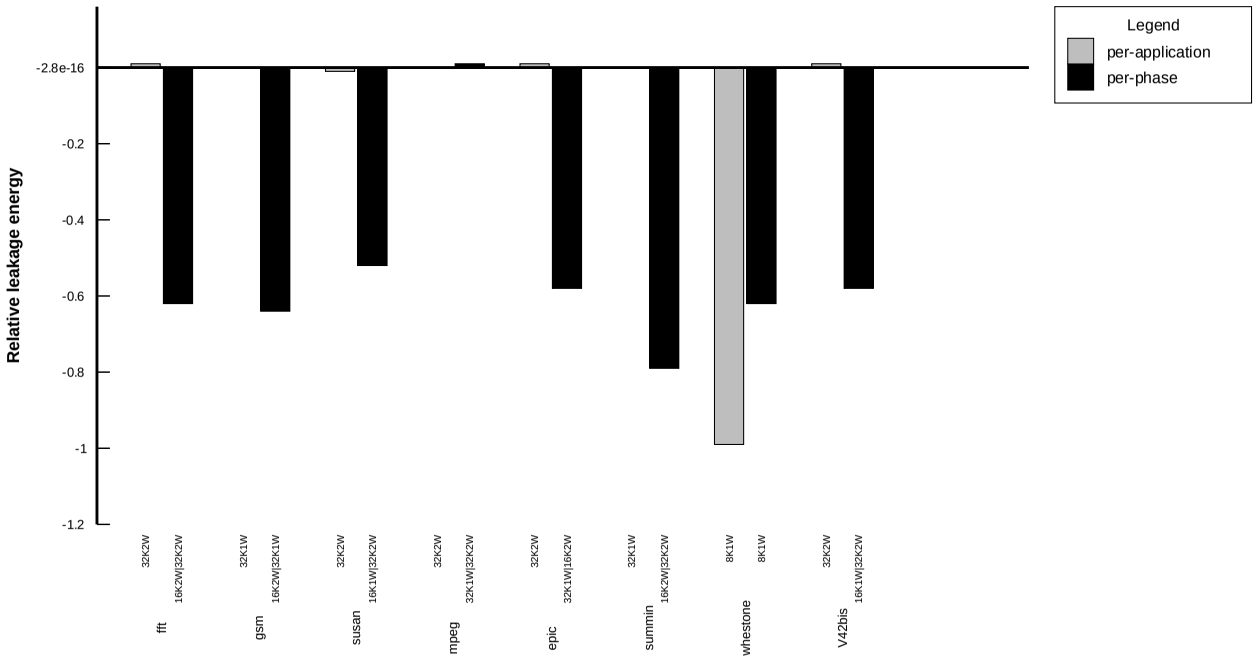

Figure 8: Relative leakage energy compared with the base cache.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399