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

Gilles Pokam
IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
gpokam@irisa.fr

François Bodin
IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
bodin@irisa.fr

Abstract

Managing the energy-performance tradeoff has become a major challenge with embedded systems. The cache hierarchy is a typical example where this tradeoff plays a central role. With the increasing level of integration density, a cache can feature millions of transistors, consuming a significant portion of the energy. At the same time however, a cache also permits to significantly improve performance. Configurable caches are becoming the "de-facto" solution to deal efficiently with these issues. Such caches are equipped with artifacts that enable one to resize it dynamically. With regard to 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.

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 the typical example of such a power/performance tradeoff design point. On one hand, a large cache allows to maintain an important fraction of the embedded code and data workload on-chip, thus reducing the amount of memory traffic and thereby improving the performance and power consumption. On the other hand, however, typical cache memory accounts for up to 80% of the total transistor count and for about 50% of the total chip area [8], making the cache memory subsystem an important source of power dissipation.

Recent researches on this area have focused on the design of configurable caches [7, 12, 1, 21, 2, 20]. 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 [7, 12, 21] have only considered configuration on a per-application basis. A drawback with this approach is that an optimal cache size, viewed from a performance standpoint, has not yet been shown to exist, whereas each application simply exhibits varying dynamic cache behaviors [16, 18]. Moreover, in the context of embedded systems, compilers play a central role in obtaining good performance; it is thus important to consider configuration schemes that improve compiler's effectiveness as well.

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. With this model, we 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. The proposed 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.

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 reconfigurable cache architecture. The compilation support to our hybrid cache model is presented in Section 4. Experimental results are presented in Section 5. Section 6 discusses related work 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 permit a cache memory system to adapt to the cache size requirement of a running program. The

various proposals of configurable cache architectures principally differ in their resizing granularity and their design complexity.

In [1], Albonesi proposes to partition a set-associative cache along its tag and data ways. Energy can be saved by allowing cache ways to be disabled/enabled on demand, according to the cache size requirement of the application. The hardware implementation is simple, with only a software register mask that enables/disables cache ways. However, this approach can only be accommodated to set-associative caches. The configurable cache design proposed in [13] is somewhat similar to selective-ways. However, instead of disabling the unused cache sections, the authors suggest to transfer useful tasks to them (e.g. instruction reuse for media processing).

Other approaches of configurable caches consist in partitioning a cache along its sets [20] or at the granularity of the cache line [10, 23]. In contrast to [1], these approaches can be accommodated to direct-mapped caches as well. However, the required implementation cost can be much more expensive. For instance, resizing a cache to the smallest and/or largest addressable number of sets with [20], requires to maintain a number of tag bits that often exceeds the one found with a conventional cache of equal size[1].

A more recent work by Zhang et al. [21] 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 degree of associativity. The proposed configuration scheme exploits a technique called "way-concatenation" which permits cache ways to be merged, while still retaining the full cache capacity but with reduced set-associativity. This approach reduces the dynamic energy since, with same cache size, lower associativity caches perform fewer switching activities than higher associativity caches. In addition, the implementation cost has been shown to be minimal.

3. Cache model description

This section describes the architectural model of the reconfigurable cache assumed for this study. While the proposed model can be applied to both data and instruction caches, only the data cache is considered.

3.1. Motivation

The motivation behind the proposed model is to emphasize the most critical application-specific cache architectural tradeoffs involved during program execution. To do so, we consider a cache with a fixed line size and modulus mapping function. Our main observation is that the performance of such a cache is mainly dictated by its size and degree of associativity. Therefore, from a software perspective, we would like to select the configuration with the lower energy consumption that minimizes the miss ratio (i.e. the one with lower degree of associativity and smaller size). 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.

[1]e.g. to upsize the number of sets from 256 to 1024 in a 32K 4-way cache with 8 tag index bits, 10 tag bits must be maintained instead.

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

Table 1. Possible cache size granularities for a 4-way, 32B line base cache with 8K per bank.

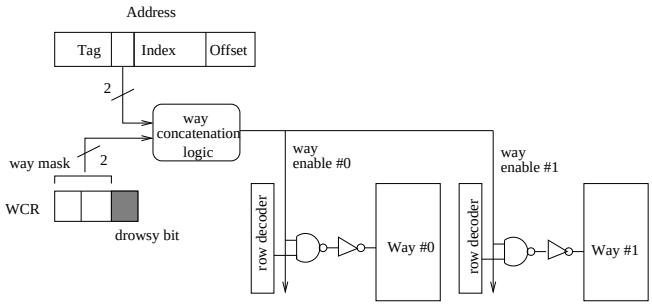

Figure 1. Baseline architecture of a 2-way associative cache.

The idea is to use a combination of schemes that permits to reconfigure a cache along its size and associativity in order to provide both in one. We do this by exploiting the variability in the cache size and the degree of associativity provided by combining the selective-way scheme [1] and the way-concatenation technique [21]. Such a hybrid scheme can provide fine-grain cache sizes at various degrees of associativity. Table 1 shows a subset of some possible cache configurations that can be exposed to the compiler. For example, starting from a 4-way 32K baseline cache configuration, 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 [21] and extends it in order to include a flexible selective-way scheme to resize a cache at runtime. Basically, with the way-concatenation scheme, one can select the number of cache ways m that can be activated on each cache lookup. In this scheme, each selected way is virtu-

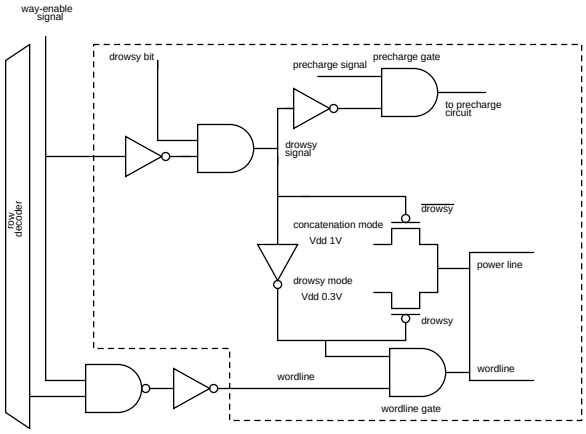

Figure 2. Drowsy cache line circuit.

ally 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 to set the number of active ways m. A way concatenation logic is in charge of carrying the active/inactive way-enable signal to each of the n cache ways. The baseline architecture is depicted in Figure 1. In this figure, the two high-order bits of the way concatenation register (WCR) are used as configuration register to fix the number of active cache ways.

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 minimizing 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.

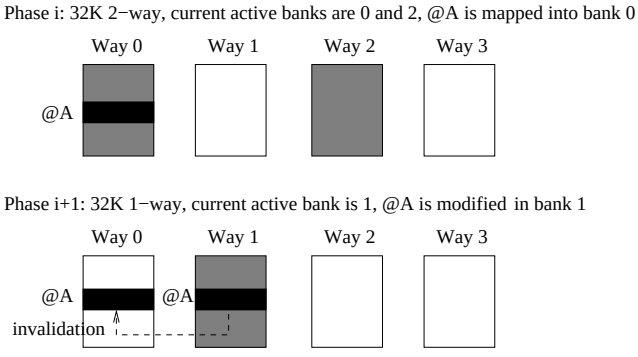

Figure 3. Reconfiguration scenario.

A possible way to overcome the cache coherency problem illustrated above is to maintain the tag and status arrays always accessible. This implies that only the data array is 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 as 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. For sake of simplicity, we assume a write-through cache policy. This guarantees data coherency whenever a cache line is provided from the upper level memory hierarchy. 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 [5], 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). We assume for this mode that the machine supply voltage can be dynamically scaled to higher values of the threshold voltage V_T. As in the gated-Vdd scheme [14], this mode reduces the leakage energy by using higher threshold supply voltages that cause the leakage current to be reduced as a by-side of the short-channel effects. In contrast to the gated-Vdd scheme, however, the supply voltage is scaled in such a way that the state of the memory cell is preserved. 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 in order to avoid the one cycle wake-up delay needed to bring a tag-way out of the drowsy mode with each write access. The fact of continuously 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 of our base cache.

Figure 2 reflects the changes introduced into the cache line in order to accommodate the drowsy mode. The drowsy bit is ANDed with the way-enable signal of each cache line. An entire cache way may 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 for each cache line switches to the lower voltage, putting the entire bank into drowsy mode. With the other cases, the supply voltage for 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 MOVWCR instruction.

3.4. Design cost

To drive the drowsy signal of each cache line, an inverter and a AND gate have been added. 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 [5], 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 [5], 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 for the presented model 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

Using the model presented previously, we describe in this section 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 optimizing each embedded application intensively. In such case, the time required to profile and pre-process the embedded applications can be justified.

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 some 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 x permits 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 [9]. We assume for the rest of this study that the caches we model support 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. We define the dynamic energy per sample interval as follows:

$$\begin{aligned} E_{\Pi_i}(map_j(x)) &= P_{\Pi_i}(map_j(x)) * E_c \\ &+ |\Pi_i| * E_T + N_{\Pi_i} * E_d \\ &+ (|\Pi_i| - Read_{\Pi_i}(map_j(x))) * E_m \end{aligned} \quad (1)$$

where

$$E_m = energy\_ratio * E_c \quad (2)$$

In (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 model 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 the memory access on a read miss event and on each write access to the cache. In (2), we estimated the $energy\_ratio$ constant to be 50.

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

Table 3. Lx microarchitecture parameters.

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

Table 4. Benchmarks used and number of accesses to data cache (in million).

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 the positions corresponding to the beginning of each phase, with each instruction initialized conveniently with the appropriate parameters (e.g drowsy bit state, way-mask). This step can be done within a separate compilation pass. Since, in this paper we focus on the potential benefits provided by such a scheme, we do not address the compilation issues associated with the automatic insertion of the reconfiguration instructions. 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.

5.1. Simulation platform and benchmarks

Our simulations were carried out on the Lx platform [4]. The Lx platform belongs to a family of customizable multicluster VLIW architectures. The implementation used in this study features a 4-issue width processor. The details of the processor microarchitecture parameters are shown in Table 3. We evaluated our cache size adaptation scheme with different applications collected from MiBench [6], Mediabench [11] and Powerstone [15] suites. All the chosen applications were compiled with the Lx native compiler, with the optimization level 3, and then run until completion. Table 4 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.

Parameter	Value
process technology	0.07 um
normal supply voltage	0.9 V
drowsy supply voltage	0.3 V
memory access latency	100 cycles
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 5. Simulation parameters.

Our simulation parameters were obtained by means of CACTI [17] and Hotleakage [22]. 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, 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 [5]. Table 5 gives an overview of the full simulation parameters that apply to this study.

5.2. Analysis of the energy and performance profiles

After empirical evaluations, we have chosen a sample interval size of $\Pi_i = 100K$ cycles to record the energy and performance values for each cache configuration, as described in Section 4.1. In order to emphasize the different energy/performance tradeoffs between the cache configurations, we graphed in Figure 4 and Figure 5 the cumulated energy values vs the number of cumulated cache misses encountered in each sample interval of program execution.

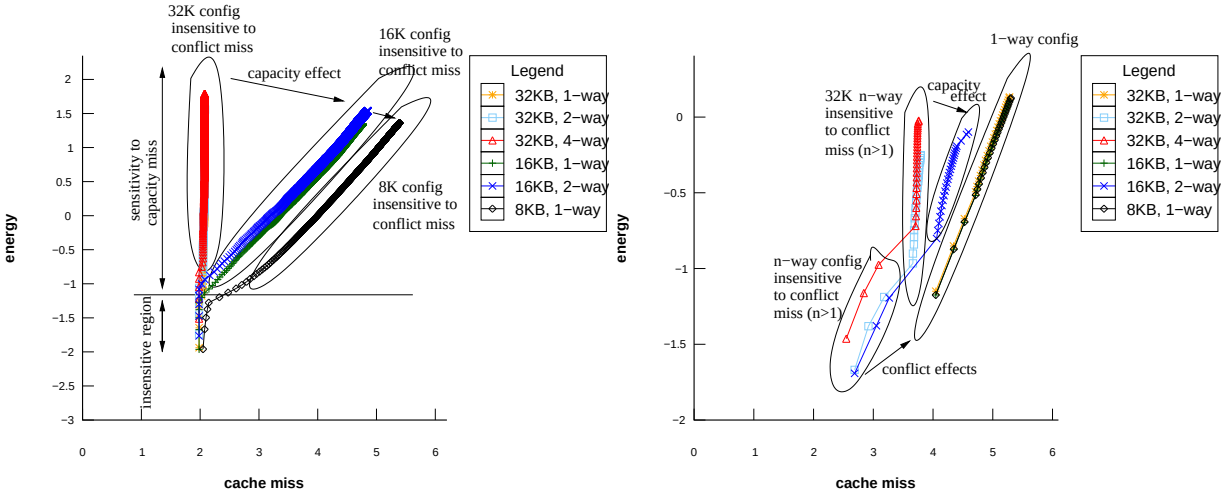

Figure 4. (a) gsm energy/performance profile; (b) fft energy/performance profile.

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 with 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 instance consider the *gsm* and *fft* applications shown in Figure 4-a and Figure 4-b, respectively.

In Figure 4-a, we can observe that there exists a threshold at which the different cache configurations are clustered according to their size, independently of the degree of associativity. In particular, we can distinguish three clusters: one with 32K configurations, another with 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, each cache configuration distinguishes itself from the other by the dynamic energy consumption due to the cache hits since the miss ratio is nearly the same. In such a case, the amount of dissipated energy is tightly coupled to the architecture of the 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 4-b, we can also observe that two main cache clusters can be distinguished: one including the n-way cache configurations with $n > 1$ and the other including the 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 with 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 5. 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. This property is exploited in the next section to construct regions of different energy/performance tradeoffs.

5.3. Working set partitioning algorithm

The objectives of achieving a partitioning of the application's working set into clusters of cache configurations may be mainly motivated by two facts. First, there is 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, which is unacceptable especially if too much unnecessary reconfigurations take place (e.g. each one for each sample point). Finally, 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:

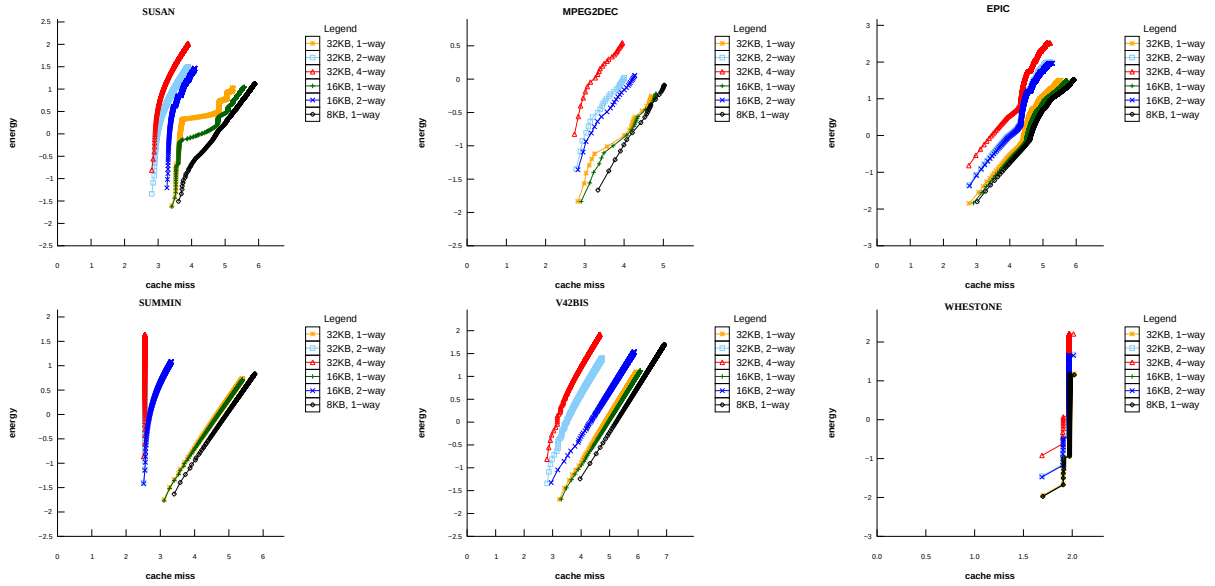

Figure 5. Energy and performance profiles

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

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

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 use a Manhattan distance vector V_k, as follows:

$$V_k = (v_1^k, v_2^k, ..., v_N^k), \quad (5)$$

where

$$v_i^k = |P_i^k - Pbase_i| + |E_i^k - Ebase_i| \quad (6)$$

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:

$$|v_i^{k2} - v_i^{k1}| < \tau \quad (7)$$

τ is a threshold value that is used to decide when to cluster or not. 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 ($i = N$). 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 6 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. The best cache configuration can be configured once before application's execution. Therefore, this configuration models the approach proposed by Zhang [21]. 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 (*gsm*, *susan*, *summin*, *mpeg*, *epic* and *v42bis*), we observe that they correspond, to some extent,

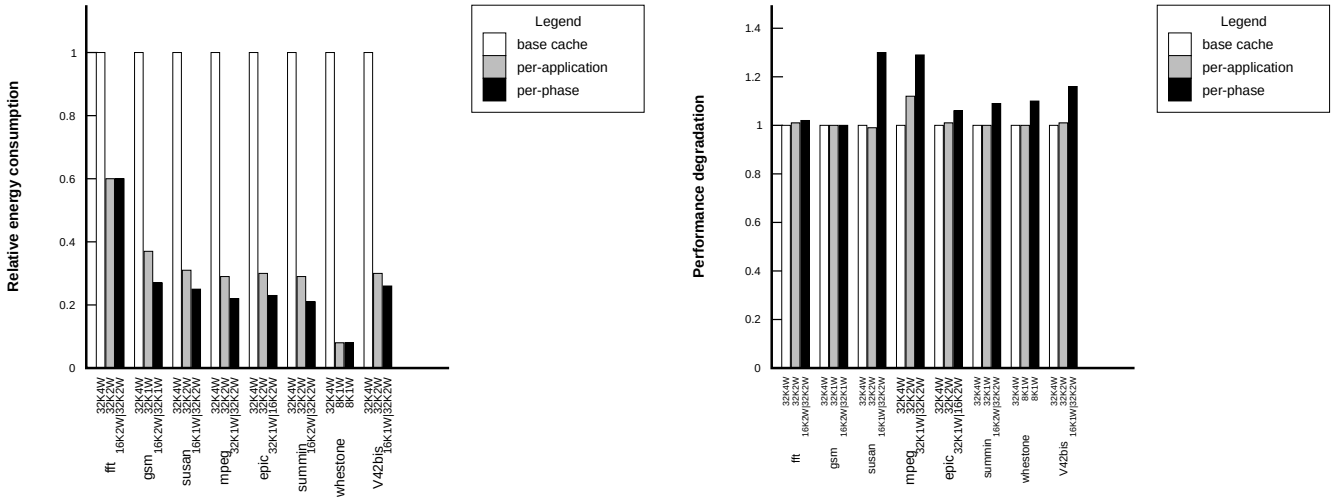

Figure 6. (left) Dynamic energy consumption; (right) Performance degradation.

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 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 7 illustrates the leakage energy of the cache configurations show in Figure 6, 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 [22]. 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 6. 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 degradations 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 [2, 19, 3] bear some similarities with our own. These researches

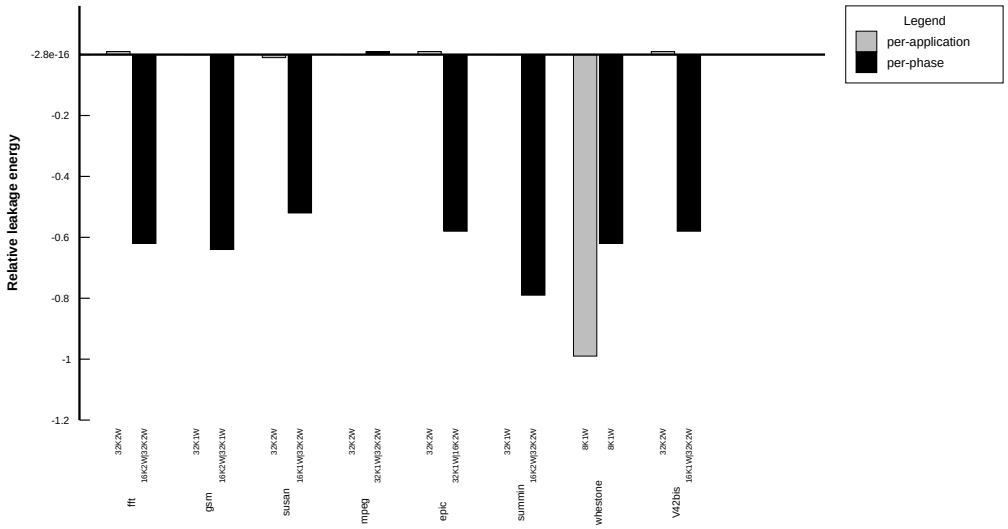

Figure 7. Relative leakage energy compared with the base cache.

share the particularity that some means of hardware adaptation scheme is required to allow a search of the optimal solution. In [2], 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 [1] 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 be used as well in the context of embedded systems, provided that some means of dynamic performance monitoring is available. In the same order, [3] reformulates the cache resizing adaptivity algorithm of [2] to use instead working set signatures, to capture phase changes and to estimate the size of a working set. This solution however also requires an extra hardware effort. Yang et al. [19] proposed a work similar to ours. They rely on the cache resizing schemes of [1] and [20] 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. [21] 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 are 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 [14] 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 [18], requiring even more aggressive, but at low cost, cache resizing architectures. For 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 with 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 that save more energy.

We envision in the future to develop a compiler technique for automatic insertion of the reconfiguration instructions. In addition, we are experiencing with more aggressive cache bank access policy to reduce the penalty due to drowsy transitions. We believe that reducing this performance penalty may make this scheme very attractive for embedded systems.

References

[1] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proc. of the 32nd Int'l Symp. on Microarchitecture*, Nov. 1999.

[2] Balasubramonian, R., Albonesi, D.H., Buyuktosunoglu, A., and Dwarkadas, S. Memory hierarchy reconfiguration for energy and performance in general purpose processor architectures. In *Proc. of the 33th Int'l Conf. on Microarchitecture*, pages 245–257, Dec. 2000.

[3] Dhodapkar, A., Smith, J.E. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, May 2002.

[4] Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., and Homewood, F. Lx: A technology platform for customizable vliw embedded processing. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, June 2000.

[5] Flautner, K., Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, May 2002.

[6] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. of the 4th IEEE Int'l Workshop on Workload Characterization*, pages 3–14, Dec. 2001.

[7] Hayakawa, F., Okano, H., and Suga, A. An eight-way vliw embedded multimedia processor with advanced cache mechanism. In *Proc. of the Third IEEE Asia-Pacific Conf. on ASICs*, Aug. 2002.

[8] J. Hennessy. The future of systems research. *IEEE Computer*, pages 27–33, Aug. 1999.

[9] Hill, M.D., Smith, A.J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38:1612–1630, Dec. 1989.

[10] Kaxiras, S., Hu, Z., and Martonosi, M. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, Jul. 2001.

[11] Lee, C., Potkonjak, M., and Mangione-Smith, W.H. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Int'l Symp. on Microarchitecture*, Dec. 1997.

[12] Malik, A., Moyer, B., and Cermak, D. A low power unified cache architecture providing power and performance flexibility. In *Proc. of Int'l Symp. on Low Power Electronics and Design*, pages 241–243, 2000.

[13] Parthasarathy Ranganathan, N.P.J., Adve, S., and Jouppi, N.P. Reconfigurable caches and their application to media processing. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, June 2000.

[14] Powell, M.D., Yang, S-H., Falsafi, B., Roy, K., and Vijaykumar, T. Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. of the Int'l Symp. on Low Power Electronics and Design*, 2000.

[15] Scott, J., Lea Hwang Lee, John Arends and William Moyer. Designing the low-power m.core architecture. In *Proc. of Power Driven Microarchitecture*, June 1998.

[16] Sherwood, T., Calder, B. Time varying behavior of programs. Technical Report CS99-630, University of California, San Diego, Aug. 1999.

[17] Shivakumar, P., Jouppi, N. Cacti 3.0: An integrated cache timing power, and area model. Technical report, DEC Western research Lab, 2002.

[18] Slingerland, N., Smith, A.J. Cache performance for multimedia applications. In *Proc. of Int'l Conf. on Supercomputing*, pages 204–217, June 2001.

[19] Yang, S-H., Powell, M.D., Falsafi, B., and Vijaykumar, T. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, Feb. 2002.

[20] Yang, S-H., Powell, M.D., Falsafi, B., Roy, K., and Vijaykumar, T. An integrated circuit/architecture approach to reducing leakage in deep-submicron high performance i-caches. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, Jan. 2001.

[21] Zhang, C., Vahid, F., and Najjar, W. A highly configurable cache architecture for embedded systems. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, June 2003.

[22] Zhang, Y., Parikh, D., Sankaranarayanan, K., Skadron, K., and Stan, M. 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.

[23] Zhou, H., Toburen, M.C., Rotenberg, E., and Conte, T.M. Adaptive mode-control: A static-power-efficient cache design. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2001.