Dynamic Cache Way Allocation for Static and Dynamic Power Reduction

Luong Dinh Hung,† Chitaka Iwama,† Niko Demus Barli,† Naoya Hattori,† Shuichi Sakai† and Hidehiko Tanaka†

1. Introduction

Power consumption has increasingly becoming a main constraint in microprocessor designs. Until recently, low-power techniques mainly focused on the reduction of dynamic power consumption. However, due to the rapid growth in magnitude of leakage current, not only dynamic power but also static power need to be reduced.

We propose way-variable caches in which some ways of the set-associative cache can be disabled during periods of modest cache requirement, while more cache ways may remain operational for more cache-intensive periods. Power consumption reduction is achieved through avoiding unnecessary switching activity (dynamic power) and suppressing leakage current (static power) in the disabled ways. We then propose an original algorithm considering temporal locality of the cache accesses to make proper way controlling decisions.

2. Way-variable Caches

2.1 Circuit Techniques

In modern designs, large on-chip caches are often partitioned into multiple subarrays to reduce the access latency. Here, we assume an associative cache structure in which each cache way locates to separate subarrays. Each subarray has its own decoder, precharger and sense-amplifier so that the subarray can be controlled independently.

Figure 1 illustrates the structure of a cache subarray. We assume that this array belongs to way0 of the cache. The subarray could be the tag portion or the data portion of the way. There is a way0_enable signal that goes to the decoder, precharger and sense-amplifier of the subarray. A large sleep pmos transistor connects the voltage supply (Vdd) to the cell array and also is controlled by way0_enable signal. The cache controller manages the current states of ways and signifies way0_enable high or low depending on whether way0 is enabled or disabled.

The circuit organization described here is quite similar to the one found in [3]. Because the purpose is to reduce dynamic power consumption in [3], gating the decoder, precharger and sense-amplifier to prevent switching activity in the disabled ways is sufficient to that purpose. Our originality is that we add the pmos sleep transistor in the supply path to cells in subarray.

When the way0 is disabled, the sleep transistor is turned off, gating all cells in the subarray from the supply voltage. The internal nodes of each cell soon drop to some intermediate voltage levels. Because leakage current highly depends on the voltage level, the leakage inside the cell is significantly reduced. Moreover, since the prechargers do not operate in the disabled ways, the bitlines drop to low voltage level. This reduces the leakage current coming from bitlines.

When using sleep transistor in SRAM, we have two possible choices: using pmos sleep transistors to gate the Vdd paths or using nmos sleep transistors to gate the ground paths [4]. In decay cache [2], it is required to insert nmos transistors in the ground paths to suppress the leakage current coming from the bitlines. The pull-down paths, rather than pull-up paths, are considered the critical paths in SRAM access. The sleep transistor sitting in the ground path inevitably degrades the cache access latency. In our way-variable cache, since the bitlines of disabled ways eventually drop to low voltage levels, the amount of leakage coming from bitlines is small. Adding pmos sleep transistor in Vdd path to suppress the leakage inside the cells is sufficient and avoids the access latency degradation in the pulldown paths.

2.2 Resizing Algorithm

For way-variable caches to be successful, we need to develop an algorithm that can properly determine when and to which direction (enable or disable more ways) the resizing will be taken. We propose here an original algorithm considering temporal locality of the cache accesses to make proper way controlling decisions.

Set associative caches usually employ the Least-Recently-Used (LRU) algorithm to choose the candidate line whenever replacement is required. LRU algorithm is implemented by providing some state bits for each cache set. Accesses to the cache set subsequently update the corresponding state bits. Our resizing algorithm makes use of these bits. Whenever an access comes to the cache, the state bits of the corresponding set are also read. If the access is a cache hit, by examining
the contents of state bits we can determine whether the line is the LRU line or not. We count the number of accesses to LRU lines \(N_{\text{LRU accesses}}\) and the total number of cache accesses \(N_{\text{total accesses}}\) over specific execution window.

The value of \(N_{\text{LRU accesses}}\) allows us to predict how the cache missrate will vary if we disable more ways. Specifically, as we decrease the number of disabled ways by one in the next execution window, we expect the cache missrate will be increased by \(\frac{N_{\text{LRU accesses}}}{N_{\text{total accesses}}} \times 100\) percent. The prediction is made on the assumption that the characteristics of cache accesses remain relatively stable over the period that quite longer than the length of single execution window. The same access that hit a LRU line in the previous window likely experiences a miss on the following windows; the line has already been evicted due to fewer enabled ways. Our resizing algorithm incrementally disable cache ways as long as the \(\frac{N_{\text{LRU accesses}}}{N_{\text{total accesses}}}\) is smaller than a predefined \(\Delta_1\) value.

It is necessary to restore the disabled ways at the execution windows with intensive cache demands. The ratio \(\frac{N_{\text{LRU accesses}}}{N_{\text{total accesses}}}\), however, does not enable us to predict quantitatively how cache missrate change when we enable more ways. Here we adopt a simple, yet effective approach that to enable all the cache ways when the cache missrate increases more than predefined \(\Delta_2\) value. The proposed resizing algorithm is shown in Figure 2.

```
//for each execution window
if(N_{\text{LRU accesses}}/N_{\text{total accesses}} < \Delta_1) //cond.1
   && enabled\_way > 1)
   enabled\_way = enabled\_way-1;
else if(current\_missrate-last\_missrate>\Delta_2) //cond.2
   && enabled\_way < max\_way)
   enabled\_way = max\_way;
last\_missrate = current\_missrate;
```

**Fig. 2** Resizing algorithm

Although not indicated in the figure, to prevent unnecessary cache reconfigurations due to transient changes, we only reconfigure the cache when the condition \(\text{cond.1 or cond.2}\) in Figure 2 is satisfied for some, say three, consecutive execution windows.

The \(\Delta_1\) and \(\Delta_2\) parameters control the “aggressiveness” of the resizing algorithm. Small \(\Delta_1\) and small \(\Delta_2\) make the algorithm “conservative” in the sense that they bias the direction that enables more cache ways. On the other hand, algorithm with large \(\Delta_1\) and large \(\Delta_2\) has the strong tendency of trading performance for more disabled ways.

Due to its simplicity, the proposed algorithm could be implemented with very modest hardware.

### 3. Evaluation

#### 3.1 Performance Evaluation

We carried out simulations to investigate effectiveness of the proposed resizing algorithm. Simulations were done using SimpleScalar simulator. We modeled a six-stage superscalar processor, having 16 KB, 32 B line, four-way set associative instruction cache and data cache, and a 256 KB, 64 B line, four-way L2 cache. L1 cache hit, miss latency were set to one and six clock cycles respectively. Simulation were done for SPECCPU2000 benchmarks. For each application, we skipped the first one billion instructions and simulated the next four billion instructions. The execution window size was set to one million instructions. \(\Delta_1\) and \(\Delta_2\) parameters were set to 0.3% and 1.0% respectively. The overhead of each cache reconfiguration was assumed to be five cycles.

**Fig. 3** Performance degradation when the number of enabled ways of instruction cache is statically varied from one to four

Figure 3 shows the performance degradation incurred for each benchmark as the number of enabled ways of the instruction cache is statically varied. The amount performance degradation is quite different among applications. Performance degradations of applications like mcf, ammp, art, swim are very small. We can execute these applications from beginning to end using the instruction cache with minimum number of enabled ways. On the other hands, performances of crafty, con, vortex, equake, mesa are very sensitive to the number of enabled ways of instruction cache. Performances tend to degrade abruptly as we disable more ways. These results suggest that we need dynamic approaches that monitor cache requirements during the lifetimes of applications and allocate appropriate number of ways accordingly.

**Fig. 4** Way-variable instruction cache: average enabled way & performance degradation

Figures 4 shows the average number of enabled ways as well as performance degradation of the way-variable instruction cache applying the proposed resizing algorithm. With our resizing algorithm, applications are executed with quite reasonable values of enabled ways. Applications whose performances are not sensitive to cache sizes (eg. mcf, ammp, art, swim) are executed with minimum number of enabled ways.
ways. On the other hand, for applications whose performances are known to be quite sensitive to cache size (eg. crafty, twolf, vortex), almost full cache ways are enabled for execution. On average, we can disable 1.83 ways out of four ways with only 1.46% IPC degradation. Referring back to Figure 3, simply changing 16 KB, 4-way cache to 8 KB, 2-way cache degraded performance by 7% on average and 28% for the worst case (crafty). The results confirm the effectiveness of the proposed resizing algorithm.

### 3.2 Power Evaluation

**Static Power.** We carried out SPICE simulations to measure the amount of leakage current exhibited by individual SRAM cells. Each cell, except the Normal cell, applies specific leakage reduction technique, described as follows.

- **Normal:** normal cache
- **DVS:** using Dynamic Voltage Scaling at cache line granularity [1].
- **NSleep-Fine:** using nmos sleep transistors at cache line granularity [2].
- **PSleep-Coarse:** using pmos sleep transistors at subarray granularity, as in way-variable caches.

Parameters at 65 nm process technology obtained from Berkeley Predictive Technology Model were used for the evaluation. Transistor threshold voltage is $|V_{th}|=0.2$ V. Vdd is set to 1 V. In case of DVS cell, Vdd in standby mode is 0.5 V.

We measured the total leakage current of the cells when they are in active state and when they are put in standby mode (low leakage mode). The results are shown in Figure 5-(a) and 5-(b) respectively. The leakage in the figure is further broken down into subthreshold leakage and gate leakage, which are the two most dominant leakage components.

![Leakage in active mode and standby mode](image)

The active leakage values are almost equal (95 nA) for all cells except the NSleep-Fine cell which is 20 nA larger. The excessive leakage is caused by the nmos sleep transistor. This transistor is on when the cell is enabled and the gate leakage of nmos in on state is significant. The PSleep-Coarse cell, which uses pmos instead of nmos sleep transistor, shows almost no increase in active leakage.

The standby leakage of DVS cell is 60 nA. While gate leakage is reduced significantly when Vdd dropped from 1 V to 0.5 V in DVS cell, subthreshold leakage is only slightly reduced. It is because the bitlines are still kept at 1 V and the subthreshold leakage current from bitlines are almost unchanged. The NSleep-Fine cell, due to its capability of suppressing the leakage paths from bitlines, offers higher degree of standby leakage reduction than the DVS cell. The standby leakage in this case is 27 nA. PSleep-Coarse cell exhibits smallest standby leakage (11 nA), which is ten times smaller than the leakage of the Normal cell.

**Dynamic Power.** The dynamic power consumed in way-variable cache is roughly proportional to the number of enabled cache ways. According to result in section 3.1, 1.83 ways out of four ways of the instruction cache can be disabled on average. By crude calculation, the dynamic power consumption of the instruction cache can be reduced by 46%. The way-variable cache may introduce extra accesses to lower level cache and these accesses consume dynamic power. We verified that using way-variable instruction cache in our experiment increased the number of L2 cache accesses by 5.4% on average. Using the Cacti tool [5], we estimated that the dynamic power consumption per read access for the 16 KB instruction cache and the 256 KB L2 cache are 0.26 nJ and 0.32 nJ respectively. The energy per read access of the L2 cache is not much larger than that of the instruction cache. The overhead due to 5.4% increase in L2 cache accesses, therefore, could be considered to be small in compare to the amount of power that can be reduced in the instruction cache.

### 4. Conclusion

The proposed way-variable caches, together with the original resizing algorithm, can offer large reduction of both dynamic and static power consumption in caches. Our evaluation verified that we can disable 1.83 ways out of four ways set-associative instruction cache while paying only 1.46% IPC degradation.

Due to space limitation, only the results of instruction cache with true-LRU replacement scheme were presented in this paper. However, we have also confirmed the effectiveness of the proposed way-variable caches and the resizing algorithm against data caches as well as the more practical quasi-LRU replacement scheme [6]. This emphasizes the wide applicability and feasibility of the techniques.

### References