Program Phase Directed Dynamic Cache Way Reconfiguration for Power Efficiency

Subhasis Banerjee
Diagnostics Engineering Group
Sun Microsystems
Bangalore, INDIA
E-mail: subhasis.banerjee@sun.com

Surendra G and S. K. Nandy
CAD Laboratory
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, INDIA
E-mail: [surendra@cadl, nandy@serc].iisc.ernet.in

Abstract— Aggressive superscalar processor with deep pipeline and sophisticated speculative execution techniques is pushing the power budget to its limit. It is found that a significant portion of this power is wasted during wrong path execution and non power optimal allocation of power hungry resources. Dynamic reconfiguration of micro-architectural resources can be exploited to bring down this waste at runtime. Lack of architectural method to capture the behavior of a program at runtime makes dynamic reconfiguration a challenge. In this paper we propose a method to characterize program behavior at runtime using conflict miss pattern of a data cache, which in turn identifies different program phases in terms of cache utilization. We use this phase information to enable/disable cache ways dynamically depending on the conflict miss pattern of a program. Using a hardware tracking mechanism we ensure that the program performance (throughput in terms of IPC) does not degrade beyond a tolerable limit. Through simulation we establish that an average improvement of 32% (best case 38%) in cache power saving is achieved at the expense of less than 2% degradation in performance for SPEC-CPU and MEDIA benchmarks. The additional hardware that detects and captures the phase information is outside the critical path of the processor and does not contribute to the overall delay.

I. Introduction

Optimal resource allocation requires runtime information of a program. Performance counters and hardware profilers are used to capture such informations and an *offline* analysis helps to find out suitable architectural design options. As an alternative, *program phase* directed optimizations can be exploited to enable *on-line* reconfiguration. Different processor parameters, for instance IPC (Instruction Per Cycle), branch miss rate, power dissipation etc, exhibit periodic behavior when monitored during entire execution of a program. The observed parameter remains almost unchanged during a small time window. Hence, a program passes through different "phases" of execution which are repetitive in nature. This phenomenon, termed as "program phase" remains invariant with input data set. An example of variation of IPC, reorder buffer (ROB) occupancy and issue rate is shown in fig. 1 for *mpeg2decode* benchmark. Each point of the plot represents a metric averaged over 10,000 cycles.

In this paper we present an efficient hardware mechanism to capture the variation of phase at runtime using conflict miss pattern of a program. Conflict miss arises due to insufficient

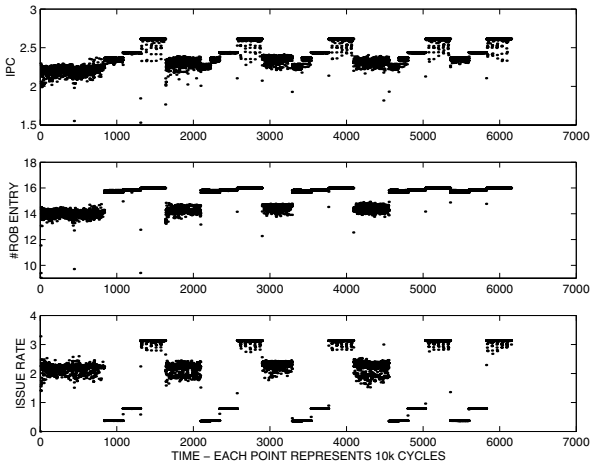

Fig. 1. *mpeg2decode* phase profile

number of ways in a cache set. As a consequence of program locality, conflict misses at each cache set generate a definite pattern when observed in a small time window. Our architecture keeps track of the number of conflict misses at each cache set during a smaller interval of program execution. At the end of every interval, the accumulated miss pattern is compared with the pattern of previous interval. A pattern matching mechanism checks for a match and classifies the pattern into different clusters which in turn characterizes a program into phases. It is found that, the number of misses do not change even with higher associativity of cache sets in certain phases of program execution. Without compromising the performance, additional cache ways can be disabled when a program enters into a phase of lower conflict misses. A feedback directed reconfigurable cache architecture incorporating our phase detector is proposed and evaluated using SPEC-CPU and MEDIA benchmarks.

II. Related Work

An analysis of basic block distribution in [12] leads to an automated approach to identify a smaller subset of code which represents overall program characteristics. An extension of the concept of Basic Block Vector (BBV) is used in [13] to identify program phases at hardware by using a BBV based phase tracker. Dhodapkar and Smith [5] use working set signature

1-4244-0630-7/07/$20.00 ©2007 IEEE.

to identify execution phases. In [13], 32 randomly chosen basic blocks are used to characterize program phase from a pool of static basic block existing in a program. The choice being random, the accuracy in detecting a phase can vary widely at different instance of simulated execution. In our work we use a miss classification table based architecture which keeps track of conflict misses from each set of a cache. Instead of taking a selective set of basic block, this scheme considers contribution from all sets of a cache to characterize a program.

Hardware detection of conflict misses is proposed by Collins and Tullsen in [4]. In this paper, we utilize the concept of conflict miss detector to identify a program phase depending on the conflict miss pattern during program execution. Albonesi proposed a scheme to allocate cache ways on demand basis in [1]. Compiler support is necessary to incorporate the scheme where a prior analysis of a particular program is done to insert cache control instructions. In [10], authors discuss two different schemes of partitioning caches, (i) associativity based partitioning and (ii) overlapped wide-tag partitioning. In our scheme we use associativity based partitioning for reconfiguration. The proposed hardware detects program phases at runtime and the decision to reconfigure cache ways are taken dynamically.

III. Simulation Framework

Our simulation framework is modeled using Wattch [2], a power estimation tool based on SimpleScalar-3.0 [3] for Alpha and PISA instruction set architecture to simulate a dynamically scheduled superscalar processor. SPEC CPU benchmarks are simulated using ALPHA instruction set and MEDIA benchmarks [6] are simulated using PISA instruction set. We model $.13\mu m$ technology for our simulation. We consider the effect of leakage current by incorporating some part of the HotLeakage [14] tool in our simulation. Just like other architectural power analysis tools, our simulation platform (Wattch) estimates dynamic power using activity based power models. In our simulation we assume that a component consumes linearly scaled power based on its bit activity (similar to Wattch) in the active state and leakage power while idle.

Table I shows the parameters used for carrying out simulations. The *base configuration* parameters approximately match those of Alpha 21264 processor. We use process parameters for a $.13\mu m$ process at 1.7 GHz in all our simulations. For SPEC CPU benchmarks, we use reduced input set available from MinneSPEC [9] benchmark suite. All the benchmarks are run to completion.

TABLE I
Processor parameters

Parameter	value
Fetch queue/RUU/LSQ size	8/64/32 instructions
Fetch/Decode/Issue width	4/4/4 instructions/cycle
Commit width	4 instr/cycle (in-order)
Functional Units	4 int ALU 1 int mult/div 2 mem ports
Branch Predictor	Combined branch predictor Bimodal 2K table 2-Level (gshare) 1K table 10 bit history 1K chooser, 4 cycles penalty
BTB	2048 entry, 4-way
Return address stack	16-entry
L1 data cache	64K, 4-way, LRU 32B block, 1 cycle latency
L1 instruction cache	64K, 2-way, LRU 32B block, 1 cycle latency
L2 cache	Unified, 1M, 4-way, LRU 32B block, 15 cycle latency
Memory	75 cycle first chunk latency 2 cycles subsequently
TLB	128 entry itlb, 128 entry dtlb, 4-way, 30 cycle miss latency

IV. Characterization of Program Phases

In this section we describe a hardware method to characterize program execution phases at runtime. A number of experiments reveal that the program behavior is primarily control dominated. A number of processor specific metrics vary significantly over different interval of a program. It is important to identify such variation and characterize it appropriately for processor reconfiguration. In this paper, we use cache conflict misses to characterize cache utilization for a given interval and reconfigure the cache ways at runtime.

Hardware Phase Detector

A schematic representation of our hardware phase detector is shown in fig. 2. It consists of an array of 16 bit saturating counter associated with 8 bit tag field. Each set in the data cache is associated with a tag-counter pair. Each counter records the number of conflict misses in the corresponding cache set. All the counters are reset at the beginning of every interval. When a cache block is evicted from a particular set, the lower order 8 bits of the evicted tag are stored in the tag field of the phase detector. A conflict miss is recorded if the lower order 8 bits of the tag of the address of a subsequent miss in the same set matches with the tag of the phase detector. Effectively the phase detector captures a subset of conflict misses as it checks for only one previously evicted tag from a cache set. The most recently evicted tag from a cache set is stored in the tag of the phase detector and compared with a miss address to detect a conflict miss. The number of bits used to store the tag field is chosen to be 8 as it is sufficient to detect around 90% of both capacity and conflict misses [4].

Every time a miss is detected by the detector, corresponding counter is increased by one. The value of the counters effectively form a vector of dimension S, where S is the number of sets in the cache. At the end of every interval a normalized vector of dimension S is formed. The duration of the interval for our experiment is taken to be 2 million instructions. An online clustering algorithm, as described in Algorithm 1, is used to classify the vector into separable clusters. Each cluster represents an execution phase of the program. A Phase History Table (PHT) records the cluster (phase) information by assigning an unique phase id for each cluster along with the phase

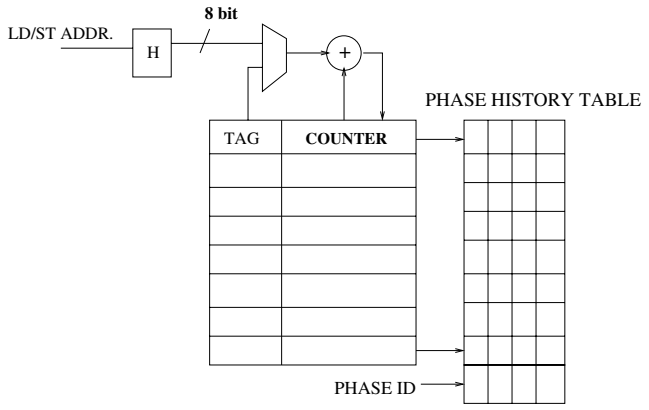

Fig. 2. Phase Detector

vector. Each entry in the PHT consists of a normalized vector and a phase identity field. An element of the phase vector is a normalized value and we observe that 4 bits are sufficient to store the value with required accuracy. A total of 8 phase id can be stored in PHT. In case of a replacement, oldest phase vector is replaced by a new one. The number of such replacements are very few in our experiment as most of the benchmarks exhibit 8 or less number of phases. As PHT is accessed once at the end of an interval (2 million instructions), the power contribution is negligible compared to the total dynamic power. For our experiment with 512 cache sets, the size of the page table is approximately $512 \times 4 \times 8$ bits = 2kB.

Algorithm 1 Detection of Phases

```
1:  get mc_vector[m];
2:  min = BIGNUM;
3:  for all i = 1 to m − 1 do
4:      dist = distance(mc_vector[i], mc_vector[m]);
5:      if dist ≤ min then
6:          min = dist;
7:          index = i;
8:      end if
9:  end for
10: if min ≤ threshold then
11:     update_(mc_vector[index], mc_vector[m]);
12: else
13:     add_new_mct_entry(mc_vector[m]);
14: end if
```

In Algorithm 1 a $threshold$ is assigned to differentiate between two clusters (phases). Every time a match is found between a phase vector (vector computed at a given interval) with a vector in PHT, the entry in PHT is updated by the geometric centroid of the two. $mc\_vector[m]$ is a vector obtained at any given interval m. The algorithm checks for a best match among all $m-1$ previous vectors. If a match is found, the new vector is merged with the appropriate cluster and the vector representing that cluster is updated with their geometric centroid, otherwise a new cluster is recorded. In the algorithm the variable $index$ keeps track of the best match vector. A phase recorded in PHT represents the geometric centroid of the interval vectors captured during program execution. The phase vector being normalized, distance between any two vectors $d_{ij} \leq 2$. A plot of $threshold$ with the number of cluster formed during execution is shown in fig. 3. The number of clusters varies linearly when $threshold$ assumes high values. With increasing $threshold$, clusters collapse to form fewer number of phases. In our experiment we choose the $threshold$ to be 1.1. With lower $threshold$ the number of clusters grow exponentially.

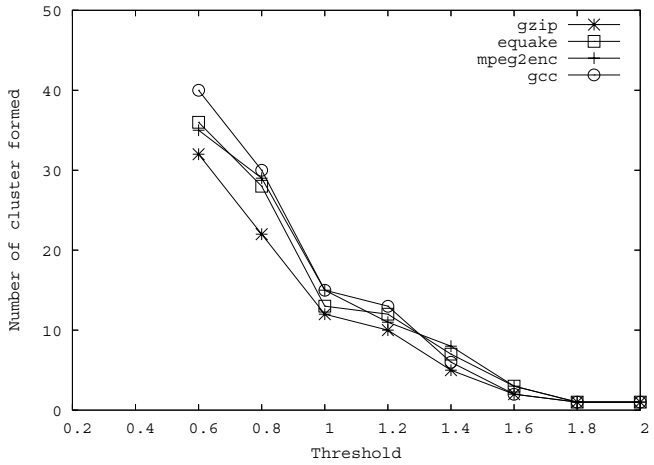

Fig. 3. Number of Cluster formed with different threshold

V. Cache Reconfiguration

In this section we describe an optimization technique to use the phase information obtained by using a hardware phase detector. Cache consumes around 20% of the processor dynamic energy. A 4 way set associative L1 data cache consumes approximately 250% more energy than a direct map cache having equal number of sets [8]. The performance gain by incorporating 3 additional ways is not more than 8-12% (best case). Moreover, a program does not need all the ways to be enabled during the entire execution time. Several methods are proposed to shutdown/disable cache ways when they are not used at its full potential. We briefly discuss a cache organization where the bitlines and wordlines are segmented for minimizing delay and propose a micro-architectural modification that helps save energy by dynamically configuring cache ways.

A. Cache Segmentation

An analytical access time model for on-chip SRAM caches is proposed and evaluated in Cacti [11]. Bit lines and word lines of a cache are segmented to improve delay at the expense of additional sense amplifier and decoder driver. Depending on number of bit line and word line segmentation, Cacti introduces six cache parameters. A cache is partitioned into two subarrays, namely data and tag subarray. The parameters N_{dwl} indicates the number of times a word line is segmented and N_{dbl} indicates the number of time a bit line is segmented in data array. N_{spd} is the number of sets that are mapped into a single section of word line. Similarly, three more parameters are assigned for the tag array, N_{twl}, N_{tbl} and N_{tspd}. Table II shows values of these organizational parameters for different cache configuration.

It is observed that the number of wordline segments of a data array is always greater than or equal to the associativity of the

cache. This indicates that a cache way spans one or more wordline segment. A wordline segment can be selectively enabled or disabled by sending control signal through cache controller. In the following section we describe a modified cache architecture to incorporate our phase detector for dynamically enabling and disabling cache ways at runtime.

TABLE II
OPTIMAL N PARAMETERS

Size	Assoc.	N_{dwl}	N_{dbl}	N_{spd}	N_{twl}	N_{tbl}	N_{tspd}
32 KB	1	1	4	1	1	4	4
32 KB	2	8	1	4	1	4	2
32 KB	4	8	1	2	1	4	1
64 KB	1	4	1	4	1	2	8
64 KB	2	8	1	4	1	4	4
64 KB	4	8	1	2	1	4	2

B. Phase Directed Reconfiguration

Fig. 4 shows a scheme to reconfigure L1 data cache ways using the phase detector and modified cache controller. A hardware counter keeps track of number of misses in the cache. A phase detector keeps track of the program phases and decides whether a particular way has to be enabled or disabled. In fig. 4, a 4-way set associative cache is shown with a modified cache controller and way select logic. The way select algorithm is described in Algorithm 2 that can be implemented as a hardware module embedded in cache controller. Algorithm 2 is invoked at the beginning of each interval. Cache controller generates a way select signal depending on the control signal from the phase detector. The way select signal enables or disables a particular cache ways by inhibiting the activation signal for precharge and sense amplifier. In the figure only two data ways and a single tag array are shown.

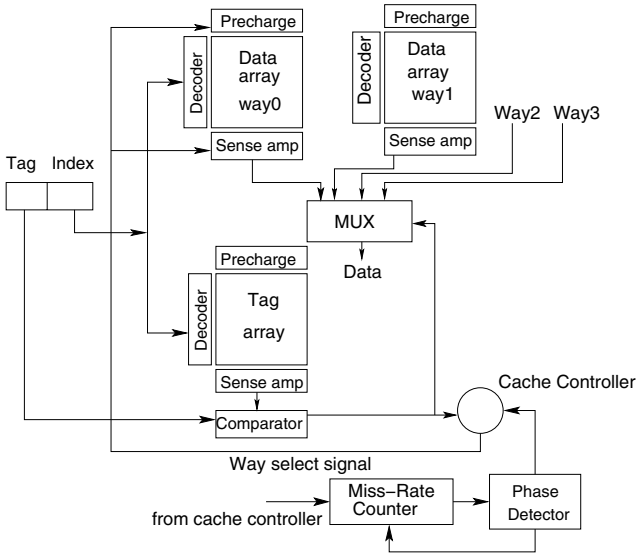

Fig. 4. Reconfigurable Cache Organization

Algorithm 2 Cache Way Selection Algorithm

```
1:  if (STATE == STABLE) then
2:    if ((present_miss − recorded_miss) < miss_noise) then
3:      miss_noise− = noise_dcr;
4:      update_state();
5:    else
6:      shutdown_one_way_of_Cache;
7:      update_state();
8:      STATE = UNSTABLE;
9:    end if
10: end if
11: if (STATE == UNSTABLE) then
12:   if (miss_rate > threshold) and (available_ways ! = 0) then
13:     enable_one_more_way;
14:     update_state();
15:   else
16:     state = STABLE;
17:     update_state();
18:     miss_noise = base_noise;
19:   end if
20: end if
```

The way select logic maintains two states, $STABLE$ and $UNSTABLE$. Every time a program enters into a new phase, the state is assigned to be $UNSTABLE$ and the cache ways are configured. After few invocation the state becomes $STABLE$. The program phase id with the way configuration is stored in a status table or in the PHT. Any subsequent detection of similar phase will enable the cache controller to reload the previous configuration corresponding to the detected phase. When cache controller loads a saved configuration while detecting a previously recorded phase, it checks whether the state is over or under-configured. If a new optimal configuration is found the previous record is overwritten by the new one. In Algorithm 2, the steps involved in $STABLE$ and $UNSTABLE$ states are described. A simple hardware performance counter keeps track of the miss rate and the information is fed back to the phase detector and the cache controller. The PHT keeps record of number of misses in each phase. Table III contains the values of different parameters used in Algorithm 2. The values are chosen appropriately by profiling all benchmarks.

TABLE III
PARAMETER VALUES FOR ALGORITHM 2

	SPEC INT/FP	MEDIA
$threshold$	3%	2%
$base\_noise$	4000	3000
$noise\_dcr$	200	100

When the cache configuration is in $STABLE$ state, it compares with the total miss count of the saved state. If the difference between the recorded state and the present state crosses a $miss\_noise$ threshold, it shuts down one cache way. This ensures that, without significant improvement in the number of misses, additional cache ways are shutdown to save power. The $update\_state()$ operation updates the PHT with the status informations containing number of misses, number of enabled ways and the phase id of the present state. To avoid frequent unwanted change from a stable to unstable state a higher value

of $base\_noise$ is set. Every time a $STABLE$ state is assigned, the $miss\_noise$ is initialized with $base\_noise$.

C. Effect of Disabling Cache Ways

When a cache way is disabled, blocks which are still valid in that disabled way should be accessible for future reference. Also the data residing in the cache block should be coherent when the disabled way is again enabled at a later stage. We have evaluated three schemes to address this issue.

- case 1: All data in the disabled way are flushed. This is the simplest way to ensure data coherence. All dirty blocks are written back to the L2 cache for a write back cache. Status of all other blocks are set as invalid. Cache flushing leads to significant performance loss.
- case 2: A fill buffer approach to move data from disabled to enabled way as proposed in [1]. In fig. 5, datapath modification to move data from enabled to disabled way is shown. We assume a performance penalty of 8 cycles for each such transfer in our simulation.
- case 3: An accessed block in a disabled way is stored in a 4 entry fully associative victim buffer [7]. Instead of moving the data to the enabled way, the referenced block is moved to the victim buffer. The replacement policy in the buffer is LRU and hence any subsequent hit in the disabled way will replace the LRU entry from the buffer. While moving data from disabled way to the victim buffer, the entry in the disabled way is invalidated.

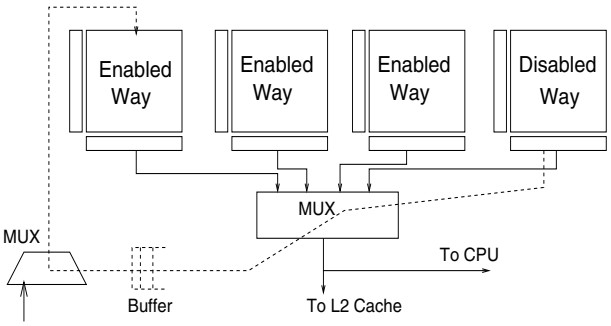

Fig. 5. Modified datapath in a 4 way Reconfigurable Cache

In fig. 6 a victim buffer is placed in the datapath. A request for a data in disabled way is moved to the victim buffer. This effectively holds the data in the buffer for a subsequent reference instead of flushing all data from disabled way at the first instant. Eviction from the victim buffer writes data back to L2 cache.

Case 3, which we incorporate with our simulation performs better than the first one where disabled ways are flushed. More than 75% of the accesses in the disabled ways are serviced by the victim buffer, effectively saving the penalty incurred during cache flushing. In fig. 7 we plot the number of misses in the disabled way that are serviced by the victim buffer. One advantage of our method of using victim buffer over the scheme as described in case 2 is that, it avoids frequent activation of

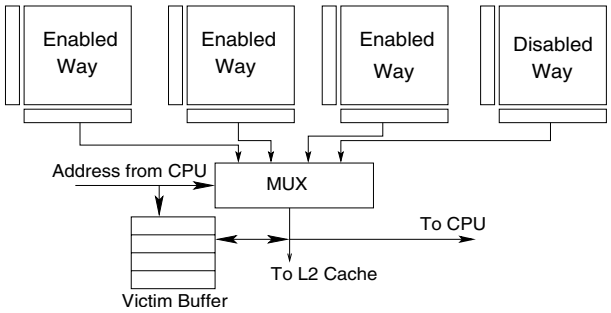

Fig. 6. 4-way Reconfigurable Cache with Victim Buffer

precharge and sense amplifier logic of the disabled way. When a data access to the disabled way is serviced by victim buffer, the latency encountered is equivalent to L1 cache latency. Every hit to victim buffer saves 7 cycles penalty compared to case 2.

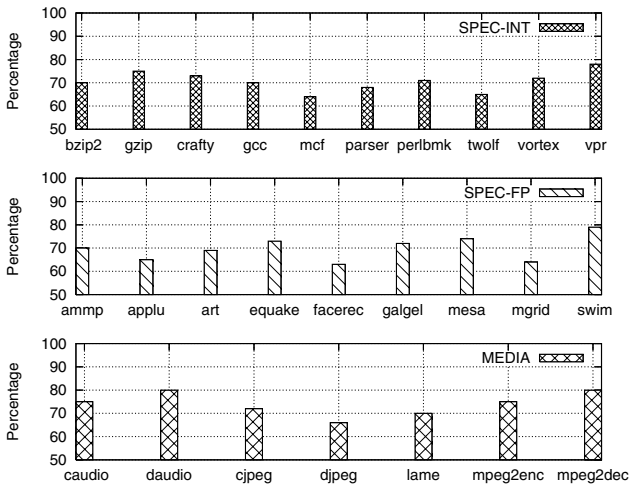

Fig. 7. Percentage of misses in disabled way served by victim buffer

D. Hardware Overhead

The hardware described in fig. 2 consists of tags associated with simple saturating counters. The number of such tag-counter pair is equal to the number of sets present in the cache. While estimating the power of the proposed hardware, the major part of dynamic power dissipation is on the 8-bit tag part of the phase detector. The counter is updated only if there is a hit in the phase detector's stored tag. The contribution for the counter is negligibly small. The idle power is estimated by the HotLeakage power model as mentioned earlier. The tag part is modeled as a direct-mapped cache tag which is indexed by the index bits of the address. On a cache miss the structure is accessed and checked for a possible hit in the tag. Our estimation shows that the power overhead averaged over all benchmarks is around 2% of the power dissipated by the cache. Power dissipated by PHT (size = 2kB) is also included in computing total power. The effects of additional modification of the datapath as described in fig. 4 are not considered in the simulation as they do not contribute significantly. The power consumed by victim

buffer in fig. 6 is modeled using the SRAM power model used in Wattch simulator.

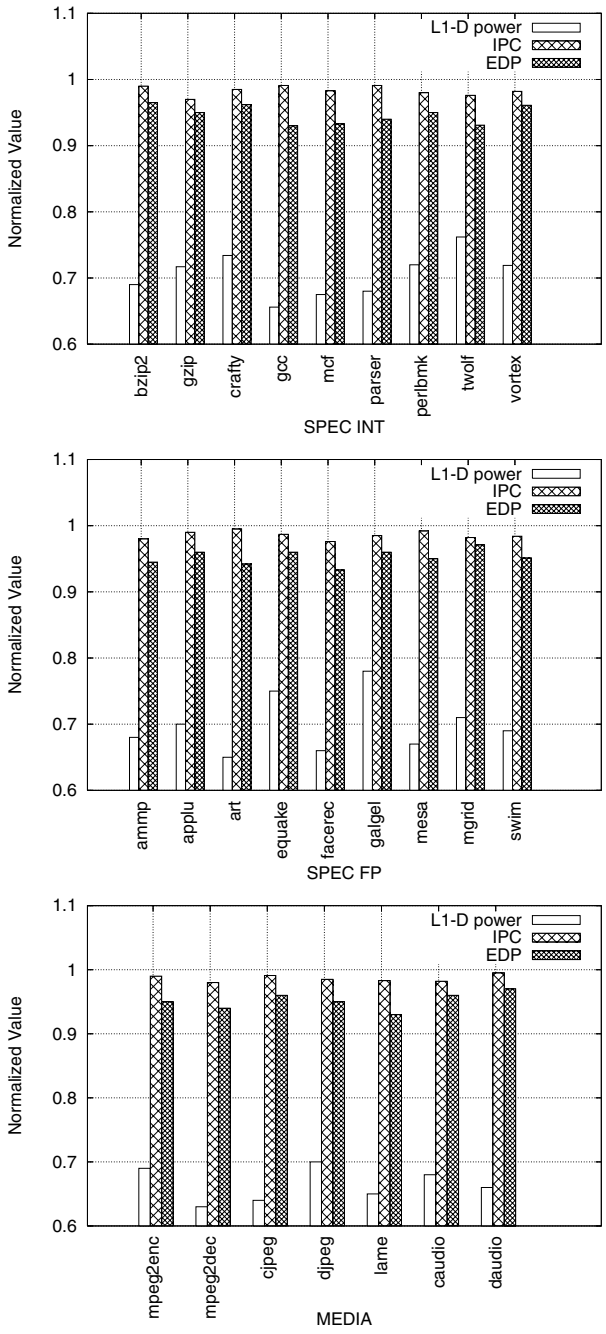

Fig. 8. Normalized L1 D-cache power, IPC and EDP for SPEC-INT, SPEC-FP and MEDIA benchmarks

VI. Power-Performance Trade-off

In our simulation we have compared our scheme with the base configuration where the number of cache ways are 4. In fig. 8 the plot of normalized L1 data cache power, IPC and total EDP are shown for SPEC-INT, SPEC-FP and MEDIA benchmarks. It shows an average saving of 32% of L1 data cache power with almost negligble loss of IPC. An improvement of 5% in EDP shows that the total energy saving is achieved offsetting the overhead of the additional hardware.

VII. Conclusion

Runtime optimization of processor resources are limited by the techniques to identify the appropriate point to invoke reconfiguration. A lossy profiling technique always inherits the risk of over-configuration or under-configuration. More sophisticated robust technique comes with the cost of complex hardware and more power consumption. In this paper we have introduced a feedback directed cache reconfiguration scheme that dynamically reconfigure a way partitioned cache. The hardware phase detector captures dynamic behavior of a program with the help of counters and associated reconfiguration logic. One disadvantage of this method is that, phases that change at smaller granularity are averaged out when reconfigured for a larger interval. We obtain 32% power benefit for L1 data cache with marginal loss of IPC for all benchmarks. The hardware reconfiguration overhead in our scheme is minimal and it does not offset the power benefit.

References

[1] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," Journal of Instruction Level Parallelism, May 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimization," International Symposium of Computer Architecture, June 2000.

[3] D. Burger, T. Austin, and S. Bennet, "Evaluating future microprocessors: The simplescalar tool set," University of Wisconsin-Madison, Technical Report CS-TR-96-1308, July 1998.

[4] J. D. Collins and D. M. Tullsen, "Hardware detection of cache conflict miss," International Symposium on Microarchitecture, 1999.

[5] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," Proceedings of the 29th annual International Symposium on Computer Architecture, 2002.

[6] M. Guthaus and et. al., "Mibench: A free, commercially representative embedded benchmark suite," IEEE 4th Annual Workshop on Workload Characterization, Dec 2001.

[7] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," Proceedings of the 17th annual international symposium on Computer Architecture, 1990.

[8] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low power caches," In Proc. of Intn'l Symp. Low Power Electronics Design, 1997.

[9] A. J. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research," Computer Architecture Letters, Volume 1, June, 2002.

[10] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," Proceedings of the 27th International Symposium on Computer Architecture, June 2000.

[11] G. Reinman and N. Jouppi, "Cacti 2.0: An integrated cache timing and power model," Western Research Laboratory, Research report 2000/7, Feb 2000.

[12] T. Sherwood, E. Perlman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," International Conference on Parallel Architectures and Compilation Techniques, September 2001.

[13] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," Proc. of the Intl. Symp. on Computer Architecture, 2003.

[14] Y. Zang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature aware model of subthreshold and gate leakage for architects," Technical Report CS 2003-2005, Department of Computer Science, University of Virginia, March 2003.