Reducing Latencies of Pipelined Cache Accesses Through Set Prediction

Aneesh Aggarwal
Electrical and Computer Engineering
Binghamton University
Binghamton, NY 13902
aneesh@binghamton.edu

Abstract

With the increasing performance gap between the processor and the memory, the importance of caches is increasing for high performance processors. However, with reducing feature sizes and increasing clock speeds, cache access latencies are increasing. Designers pipeline the cache accesses to prevent the increasing latencies from affecting the cache throughput. Nevertheless, increasing latencies can degrade the performance significantly by delaying the execution of dependent instructions.

In this paper, we investigate predicting the data cache set and the tag of the memory address as a means to reduce the effective cache access latency. In this technique, the predicted set is used to start the pipelined cache access in parallel to the memory address computation. We also propose a set-address adaptive predictor to improve the prediction accuracy of the data cache sets. Our studies found that using set prediction to reduce load-to-use latency can improve the overall performance of the processor by as much as 24%. In this paper, we also investigate techniques, such as predicting the data cache line where the data will be present, to limit the increase in cache energy consumption when using set prediction. In fact, with line prediction, the techniques in this paper consume about 15% less energy in the data cache than a decoupled-accessed cache with minimum energy consumption, while still maintaining the performance improvement. However, the overall energy consumption is about 35% more than a decoupled-accessed cache when the energy consumption in the predictor table is also considered.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*Pipeline processors, RISC architectures*

Keywords

Cache Memory, Speculative Cache Access, Set Prediction, Instructions Per Cycle, Line Prediction

ICS '05 June 20-22, Boston, MA, USA.

1 Introduction

The performance gap between the processor and the memory is ever increasing. Cache memories have long been used to bridge this performance gap. Caches are Random Access Memory (RAM) structures that store the most recently used data, and exploit the temporal and spatial locality of memory references to improve performance. Access latencies of the caches have to be small for them to be effective. However, with the CMOS scaling trends leading to faster transistors and relatively longer wire delays, having low latency cache memories may not be feasible [9], especially if larger caches are required. This is because, RAM delays scale much slower than transistor delays due to the long wires required to access the RAM structures. To prevent the increasing cache access latencies from affecting the cache throughput, the cache access is usually pipelined.

The main delay components in a data cache read access are delays in decoding the set-index, comparing the address tag with the tags in the set, reading the data, and driving the output data to the destination. Two straightforward approaches to pipeline the cache read access are shown in Figure 1[11]. In Figure 1(a), the data is read in parallel to tag comparisons. We call this type of cache access as *non-decoupled* cache access. *Non-decoupled* cache access helps in reducing the cache access latency, but increases the cache energy consumption. With increasing concern for chip power consumption [13, 20], *decoupled-accessed* caches (shown in Figure 1(b)) are becoming more popular [15]. Studies have shown that caches can consume up to 20% of the overall chip power consumption [13, 17]. *Non-decoupled* cache access results in a significant number of redundant and unnecessary data reads, driving up the cache energy consumption. In a *decoupled-accessed* cache in Figure 1(b), first the tags are compared to determine the cache line in the set containing the data, and then the data is read only from that cache line. To further pipeline the cache access, wave pipelining [9] can be used, which overlaps the wordline delay of the next access with the bitline and the sense amplifier delays of the previous access. However, even with pipelined cache access, the latency of a single cache access is not reduced (only the cache throughput is improved) and the instructions dependent on load instructions still need to wait for many cycles before they can get their operands. These delays in the execution of dependent instructions can result in significant degradation in instruction throughput.

In this paper, we propose data cache set and memory address tag prediction to reduce the effective cache access latency of pipelined cache accesses. *In this paper, we work principally with a decoupled-accessed cache of Figure 1(b). However, the techniques of this paper can also be applied to a non-decoupled-accessed cache.* Cache set prediction for instruction caches has been pro-

Figure 1. (a) Pipelined Non-Decoupled-Accessed Data Cache; and (b) Pipelined Decoupled-Accessed Data Cache Read Access

posed in [7] and implemented in [15], where the next set from which the instructions will be read is predicted based on the current set being accessed. Such a scheme may not work very well for data caches, because the same set can be accessed from various points in a program, and the next set to be accessed may depend on the point in the program from where the current set is accessed. However, instruction-wise set prediction can be very accurate for the data caches. In this technique, for each memory instruction, the set where its data can be present, is predicted. First of all, to improve the set prediction accuracy, we propose an adaptive set-address predictor. This predictor is based on our observation that for some benchmarks, predicting the set of the access is more accurate, whereas for other benchmarks, predicting the entire address of the access (from which the set can be extracted) is more accurate. In our techniques, the address tag (which is observed to be highly predictable) is also predicted. To reduce the cache access latency, once the memory instruction is issued, based on the aggressiveness of the technique, the access to the data cache is started in parallel to computing the address for the instruction, thus reducing the load-to-use latency and improving the instruction throughput. We found that using a adaptive set-address predictor for accurate set prediction, the Instructions Per Cycle (IPC) count can be improved by as much as 24% for the SPEC2K benchmarks. In this paper, we also propose techniques to limit the increase in cache energy consumption when using set prediction. Additional energy is consumed by the caches when the set is mispredicted. To reduce the additional energy consumption, we propose techniques to reduce the set misprediction rate. We also investigate a technique where the cache line (in the set) containing the data is also predicted and is first checked for the data. With line prediction, the set-prediction techniques in this paper consume about 15% less energy in the data cache than a decoupled-accessed cache, while still maintaining the IPC improvement.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents the statistics on the predictability of the memory address parameters, and also discusses a few simple predictors to predict memory address parameters. In this section, we also propose an adaptive set-address predictor for more accurate set prediction. Section 4 discusses two schemes to employ data cache set prediction to reduce the effective cache access latency. Section 5 presents the experimental setup and the results. Section 6 discusses data cache energy consumption results and presents techniques to reduce the energy consumption in the data cache when employing set prediction. We conclude in Section 7.

2 Related Work

Since the origin of caches, researchers have been trying to improve its efficiency. One approach to improve the cache efficiency is to predict different memory address parameters. Way-prediction has been proposed in [8, 18] to reduce the cache access latency and to improve the cache energy efficiency. In [8], the authors propose predicting the way (in a set) where the data is present to reduce the latency of a sequential associative cache, where instead of sequentially traversing all the ways in the set to look for the data, the predicted way is accessed first. Powell et. al. [18] use way prediction to reduce the energy consumption in the caches. However, the proposed way prediction techniques have to wait for memory address computation, and cannot be used to reduce cache access latency in current caches where all the ways in a set are accessed simultaneously. Calder et. al [7] propose both set and line prediction for instruction caches. In [7], the authors maintain a pointer into the instruction cache, indicating the target instruction of a branch, to improve the accuracy of fetch and branch prediction. As mentioned in the introduction, such a set prediction may not work for the data caches.

The entire memory address prediction has also been proposed to initiate the memory access earlier and hide the memory hierarchy latency. Lipasti et. al. [16] predict the address of a load instruction and the values are speculatively loaded, and the dependent instructions are executed based on the speculated loaded data. The concept of context-based and correlated load address predictions was proposed in [4]. Austin and Sohi [1] propose caching the stack pointer and the global pointer registers as well as a number of general-purpose registers that have been recently used as base or index components of memory address, and use these registers to speculatively generate a virtual address at the end of the decode stage and access the cache speculatively. However, in all these cases, the misspeculation penalty can be very high as a large number of instructions that have speculatively executed will have to be squashed and re-executed. In our techniques, data is not speculatively used, and the misspeculation penalty is minimal. In addition, as we show in the next section, set prediction can be much more accurate than memory address prediction, resulting in fewer misspeculations.

Various other techniques have also been proposed to reduce load-to-use latency. One such technique is data prefetching, which is used to bring the most likely used data closer to the processor. Baer and Chen defined the concept of data prefetching and introduced last-address and stride-based predictors [3, 10]. [12] propose to share the same stride-based prediction structures to perform both address prediction and data prefetching simultaneously. Another technique to reduce the load-to-use latency is to resolve the load address as early as possible. [5] safely computes the addresses of memory references in the front-end part of the processor pipeline, using register tracking. [2] proposed a fast carry-free addition of the index portions of base and offset address components to speedup address generation.

3 Memory Address Parameters Prediction

In this section, we discuss the instruction-wise predictability of different memory address parameters, such as the set to be accessed, the memory address tag, and the entire memory address, and a few simple predictors to predict these parameters.

3.1 Predictability

To measure the instruction-wise predictability of different memory address parameters, we collect statistics for repeated memory ac-

cesses from the same memory instructions. We present the statistics for only the memory read (load) instructions because of their more influence on performance. For the measurements, we maintain a 64K-entry table indexed using the PC of the load instructions, and the memory address parameters of the current load instruction is compared with the parameters of the previous instances of the same load instruction. Table size of 64K entries was found to be enough to capture almost all the active load instructions (there were almost no misses due to conflicts between different load instructions). We define six different categories of sets accesssed, based on whether the current set accessed (i) is the same as the previous set accessed, (ii) is logically 1 more than the previous 2 sets accessed (the previous 2 sets accessed were the same and the current access crosses the cache block boundary and is in the next logical set), (iii) is logically 1 less than the previous 2 sets accessed, (iv) is in continuation of a stride-based access of sets, (v) lies among the previous 2 distinct sets accessed, and (v) does not lie in any of the categories mentioned above. We use the term *logical* to indicate that the set next to the last physical set in the cache is the first physical set, and the set previous to the first physical set is the last set. The address tags are only divided into two different categories: (i) same as the last tag generated, and (ii) all the other generated tags. The rest of the categories studied for set predictability do not apply for address tags. The memory addresses are divided into four different categories: (i) same as the last address, (ii) addresses in continuation of a stride-based access, (iii) addresses among the last 2 distinct addresses, and (iv) the remaining addresses. Figure 2 shows the measurements for many of the SPEC2K benchmarks. In Figure 2, the first bar shows the statistics for set, the second bar for address tags, and the third bar for memory addresses.

Let us discuss the address tags first. Intuitively, the memory addresses accessed by multiple instances of the same load instruction will most probably change only in the lower bits with the upper bits remaining the same. This is especially true if the memory accessed by the load instruction either remains the same or increases or decreases is small strides. Figure 2 also shows that, more than 90% of the tags generated are the same as the last tag generated, and can be predicted using a *last tag predictor (LTP)*, except for `mcf`. In fact, the address tag has the highest predictability among the three different parameters studied.

The results for memory address and set predictability are much more interesting. Figure 2 shows that the stride-based predictability of memory addresses is more than that of cache sets, whereas the case is the opposite for last-value-based predictability. This is because some of the strides are lost by set prediction. For instance, consider a sequential array access where the size of each array index is one-half the cache block. In this instance, the sets accessed will not show any strides whereas the memory addresses will. In the same instance, the sets accessed will show some predictability based on the last-value, whereas the memory addresses will not. In addition, in cases where a load instruction accesses the most recently accessed set even though the memory address has changed randomly, the set may be correctly predicted using a last-value-based predictor but the memory address will be mispredicted. Overall, Figure 2 shows that, when combining the stride-based and last-value-based predictability, sets have more predictability for `gzip`, `bzip2`, `parser`, `twolf`, `vpr`, `equake`, `mesa`, `swim`, and `apsi` benchmarks, whereas addresses have more predictability for `gcc`, `mcf`, `mgrid`, `applu`, and `art` benchmarks. Note that the predictability of addresses is higher for benchmarks that have a relatively higher stride-based predictability. Figure 2 also suggests that for most of the load instructions, the different memory address parameters can be predicted by either using a last-value-based predictor or a stride-based predictor. Figure 2 shows that there are also some load instructions (especially for `gcc`, `mgrid`, and `applu`) that access a set logically next to the one accessed by the previous 2 instances of the same instruction. These accesses are mostly a result of random accesses and when the stride crosses over into the next set. There are also some instructions that access a set or an address which is among the last 2 distinct sets or addresses accessed. Our studies also showed that there are more instructions that access a set which lies in the last 4 distinct sets accessed. We may be able to predict these kind of accesses using some kind of a history pattern based predictor to further improve the prediction accuracy of the data cache set predictions. However, in this paper, we do not study pattern based predictors for data cache set predictions.

3.2 Predictions

In this section, we study a *last tag predictor (LTP)* for address tags, a *combined stride-based and last-value-based set predictor (SSP)* for sets, and a *combined stride-based and last-value-based address predictor (SAP)* for addresses. In all the predictors, the prediction is made only for load instructions and is made at dispatch time, and the predictor is updated at commit time. In the combined predictors, if a prediction cannot be made using a stride, then the prediction is made based on the last value. Since the measurements are performed in a cycle-accurate simulator, to avoid making predictions based on stale data (the predictor is updated when an instruction commits), counters are used for the *SSP* and *SAP* predictors to count the number of decoded (but not yet committed) instances of an instruction.

Figure 3 shows the prediction accuracies for the different memory address parameters, for predictor table size of 1K entries. We experimented with table sizes of 256 entries up to 64K entries, and found that for many of the benchmarks, the accuracies start to saturate at 1K-entry table size. In Figure 3, the entire set of load accesses are divided into correct, wrong and no predictions. Figure 3 shows that the *SSP* gives an average correct prediction of about 50% for the integer benchmarks (`gcc`, `gzip`, `mcf`, `bzip2`, `parser`, `twolf`, and `vpr`), and an average correct prediction of about 85% for the FP benchmarks (`equake`, `mesa`, `mgrid`, `swim`, `applu`, `apsi`, `art`, and `wupwise`). These numbers for the *LTP* are 80% and almost 100%, respectively, and for the *SAP* are 55% and 85%, respectively. The percentage of correct predictions in Figure 3 is less than the predictability (even when considering only last-value-based and stride-based predictabilities) in Figure 2, because no predictions can be made for a particular load instruction till at least one of its instances commits, and also because stale commit data results in mispredictions for some of the instructions. More importantly, correct address predictions (and hence correct set predictions when the set is extracted from the address) are more than set predictions for `gcc`, `mcf`, `mgrid`, `applu`, and `art`, whereas they are less than set predictions for the rest of the benchmarks (except for `wupwise`), as explained in the previous section. Based on this observation, we propose an *adaptive set-address predictor (ASAP)* for predicting the sets of memory accesses.

3.3 Adaptive Set-Address Predictor (ASAP)

In the previous section, we observed that the set prediction accuracies are better for some benchmarks, and the address prediction accuracies are better for the others. With a correct address prediction, the set can be extracted from it and hence would lead to a correct set prediction. This suggests that a combined set-address predictor will work better for set predictions. In this predictor, a

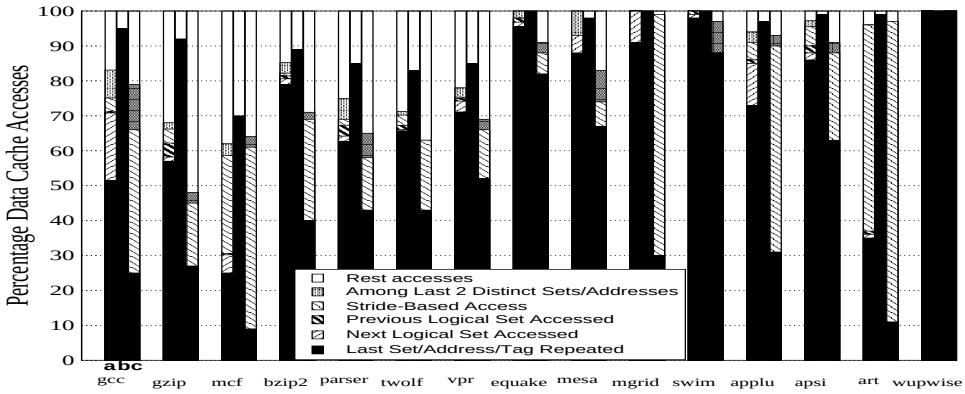

Figure 2. Predictability of (a) Data Cache Sets; (b) Memory Address Tags Generated; (c) Memory Addresses Generated

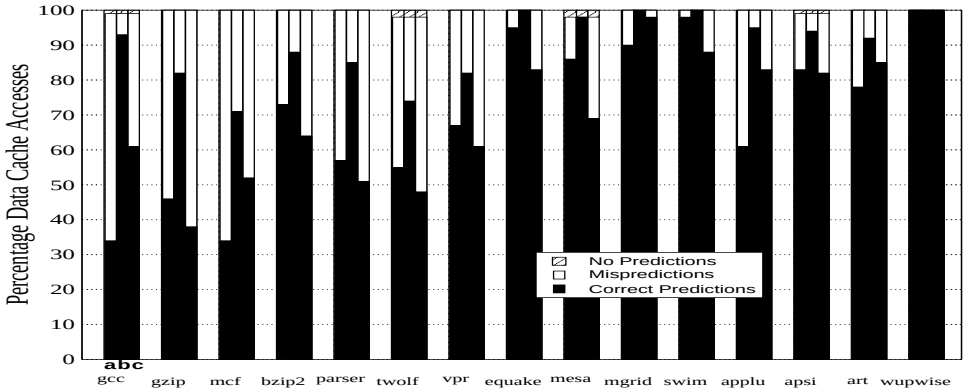

Figure 3. Prediction Accuracies for (a) Combined Stride-Based and Last-Value-Based Set Predictor (SSP); (b) Last Tag Predictor (LTP); (c) Combined Stride-Based and Last-Value-Based Address Predictor (SAP)

confidence counter is maintained each for the set and the address predictor. When a set prediction is to be made for an instruction, either set or address prediction is made, depending on whose confidence counter is higher. For instance, if the confidence counter of the address prediction is higher, then the set is extracted from the predicted address. If the confidence counters of both the predictors are the same, then the predictor can be arbitrarily chosen (we choose the address predictor because, as discussed in Section 4.2, we also extract the byte offset from the address prediction). When an instruction commits, both the set and address predictors are updated based on the actual set and address values, and the confidence counters for the set and address predictors are incremented or decremented depending on whether the predictors would have given a correct prediction for the instruction or not. In our experiments, we used 2-bit saturating confidence counters for both the set and the address predictors, and found that the set prediction accuracies obtained from the *ASAP* predictor are almost equal to the maximum prediction accuracies among the set and the address predictions in Figure 3.

Each entry in an ASAP table consists of a valid bit, the PC-tag entry, instance counter, and entries for the set and address predictors. Each of the set and the address predictors entries consist of the last value, a confidence counter, a stride, and a stride validity bit. The validity bit indicates whether the stride is tentative, or valid. The ASAP table is accessed after the decode stage and before the dispatch stage. Deep pipelines allow a pipelined access to the ASAP table, where decoding the PC value (to activate the appropriate entry in the table) can be decoupled from reading the entry. Once the entry is read, additional cycles can be expended in checking the valid bits, and comparing the PC tags and confidence counters of the predictors, followed by the predictions based on the comparisons. The predicted address tag is the same as the last address tag in the entry. Set is predicted by adding the multiplication of the instance counter and the stride with the last value (set or address), if the stride is valid, otherwise the last value is used for set prediction. Small number multiplications (we use a maximum of 3 bits for the instance counter) can be easily done using shifts and additions. Figure 4 shows the schematic diagram of how the set predictions are made using ASAP. Instance counters are incremented on each prediction and decremented on each commit or squash of the load instruction. The stride validity bits are modified depending on whether a stride is observed or not.

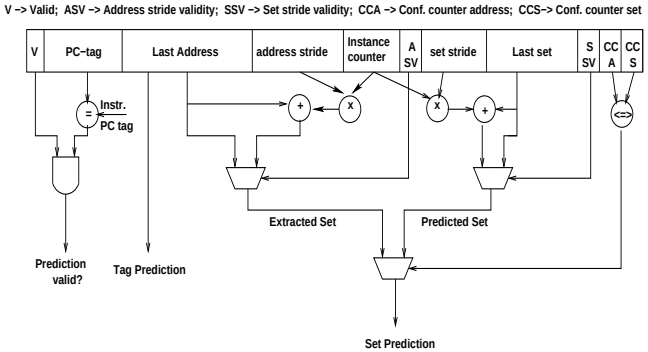

Figure 4. Schematic Set Prediction using ASAP

Next, we look at how data cache set and memory address tag predictions can be employed to reduce the cache read access latency.

4 Reducing Cache Access Latency

4.1 Basic Scheme

A pipelined decoupled cache read access is shown in Figure 5(a). Once a load instruction issues, the register files are read for the operands (we assume a 2-cycle pipelined register file access). After the address is computed and the set extracted, the set is decoded to determine the set in the cache to be accessed, followed by tag comparisons to determine whether the access is a hit or a miss. In case of a hit, the data is accessed in the next cycle which is then driven to the consuming units in the following cycle. Figure 5(b) shows the pipelined decoupled-accessed cache read access with set prediction. Once the load instruction is issued, the cache access pipeline is started in parallel to the address computation. For instance, while the load instruction reads the register file for its operands (for address computation), the predicted set is decoded, and when the address is being computed for the load instruction, tag comparisons are performed in parallel. However, for tag comparisons, the address tag from the computed address is required. There are two solutions to this problem. In most of the load instructions, the address computation only operates on the lower end bits of the address, and higher end bits (read from the register file) can be directly used for tag comparisons. We use the second solution, which is to predict the address tag. Section 3 showed that address tags are highly predictable.

Since, address computation (which is just an add operation) requires significantly less time than tag comparisons (tag comparisons require driving the tags across a large tag array), the address for the load instruction is available much before the tag comparisons finish. Hence, in the latter half of the *address computation* pipeline stage, the predictions are validated. The pipeline continues with data read on a correct prediction, as shown in Figure 5(b). In case of mispredictions, the cache access is restarted using the computed address, as shown in Figure 5(c). On a correct set prediction but a tag misprediction, the newly computed tag can be used for comparisons, saving a cycle. *However, we found very rare occurrences of the case where the set prediction is correct and the tag is mispredicted, which is also expected from the results in Section 3.* Hence, in case of any misprediction (set or tag), a new data cache read access is initiated. In this technique, on correct predictions, the effective decoupled cache read access latency reduces from 4 to 2. Note that, the cache miss penalty also reduces by 2, because the level-2 cache access can start 2 cycles early.

The instructions dependent on a load instruction are speculatively issued with the assumption that the load will hit in the cache. If the load misses in the cache, the dependent instructions are replayed [14]. With set prediction, the dependent instructions may need to be replayed not only when the load misses in the cache, but also when the set is mispredicted.

4.2 Advanced Scheme

In the *basic scheme*, once the load instruction is issued, the cache pipeline could not be started earlier because the byte offset is required for reading the data from the cache block. In this section, we investigate *byte offset prediction* to advance the *basic scheme* so that the cache access pipeline can be started earlier, further reducing the cache access latency. We again compared the prediction accuracy of a byte offset predictor against an address predictor (and extracting the byte offset out of the predicted address), and found the prediction accuracies of both the predictors to be almost equal. Hence, we still use the *ASAP* to predict the sets and the byte offsets are extracted from the address prediction part of *ASAP*, if the address predictor is used for set prediction. If the sets are predicted using the set predictor, then byte-offset prediction is not made, and the cache access pipeline is that of the *basic scheme*.

Figure 6 shows the decoupled cache read access pipeline with the *advanced scheme*. In this scheme, in parallel to register reads, the predicted set is decoded, tag comparisons are performed using the predicted address tag, and on a hit, while the address is being computed for the load instruction, the data is read using the predicted byte offset. In the address computation stage, the predictions are validated. Figure 6(a) shows the pipeline for correct predictions. In the *advanced scheme*, even if the set and the tag are correctly predicted, the byte offset may be mispredicted. In this case, only the data read operation needs to be restarted, as shown in Figure 6(b). Another option is to consider any form of misprediction as a set misprediction and to re-initiate the entire data cache access, losing the benefits obtained from the correct set and tag predictions. However, this option may keep the things simple for load replays. We take the first option in our experiments. Figure 6(c) shows the pipeline for a set misprediction. Note that, on no byte-offset prediction, the cache access pipeline is similar to the *basic scheme*. On correct predictions, in the *advanced scheme*, the effective cache read access latency reduces from 4 cycles to 1 for a decoupled-accessed cache. In this scheme also, the cache miss penalty reduces by 2 on a correct set prediction, if we wait for prediction validation before handling the cache miss.

In both the basic and the advanced schemes, if a load stalls after address computation because of ambiguous stores, the cache access pipeline with the techniques also stalls, losing the tag comparison result.

5 Performance Results

5.1 Experimental Setup

We use the SimpleScalar simulator [6], simulating a 32-bit PISA architecture. However, we significantly modify the simulator so that, instead of a single *RUU* structure working as a reorder buffer, register file, and issue queue, we have a separate reorder buffer, register file and issue queue. The hardware features and default parameters that we use are given in Table 1. For benchmarks, we use a collection of 7 integer (`gzip`, `vpr`, `mcf`, `parser`, `bzip2`, `twolf`, `gcc`), and 8 FP (`equake`, `mesa`, `mgrid`, `swim`, `wupwise`, `applu`, `apsi`, and `art`) benchmarks, using *ref* inputs, from the SPEC2K benchmark suite. The statistics are collected for 500M instructions after skipping the first 1B instructions. Even though the schemes in the paper can be applied to any pipelined cache accesses, we employ the schemes only for the Level-1 data caches. In our experiments, we use a 4-ported 1K-entry, direct-mapped ASAP table with each entry of size 80 bits which include 1 valid bit, 2 stride validity bits, 4 confidence counter bits, 20 PC-tag bits, 3 instance counter bits, 7 last stride bits, 32 last address bits, 3 stride bits for the set prediction, and 8 stride bits for the address prediction. Hence, the total size of the table is 80K bits or 10 KB. A maximum of 2 set predictions and 2 updates can be made simultaneously using the 4 ports in the table. We assume a 4-stage pipelined access to the predictor table, performed in parallel to the original pipeline.

5.2 IPC Results

In this section, we present the IPC results of our experiments. We experimented with a baseline 4-staged pipelined decoupled-accessed cache, decoupled-accessed cache using the *basic scheme*,

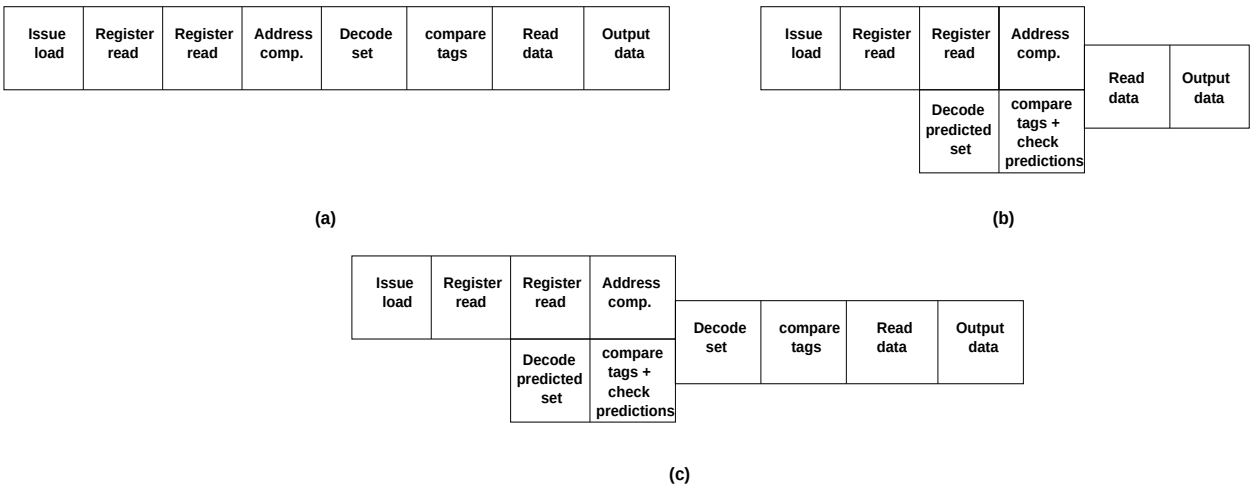

Figure 5. Pipelined Decoupled-Accessed Data Cache Read Access (a) Original; (b) With Basic Scheme on Correct Predictions; and (c) With Basic Scheme on Mispredictions

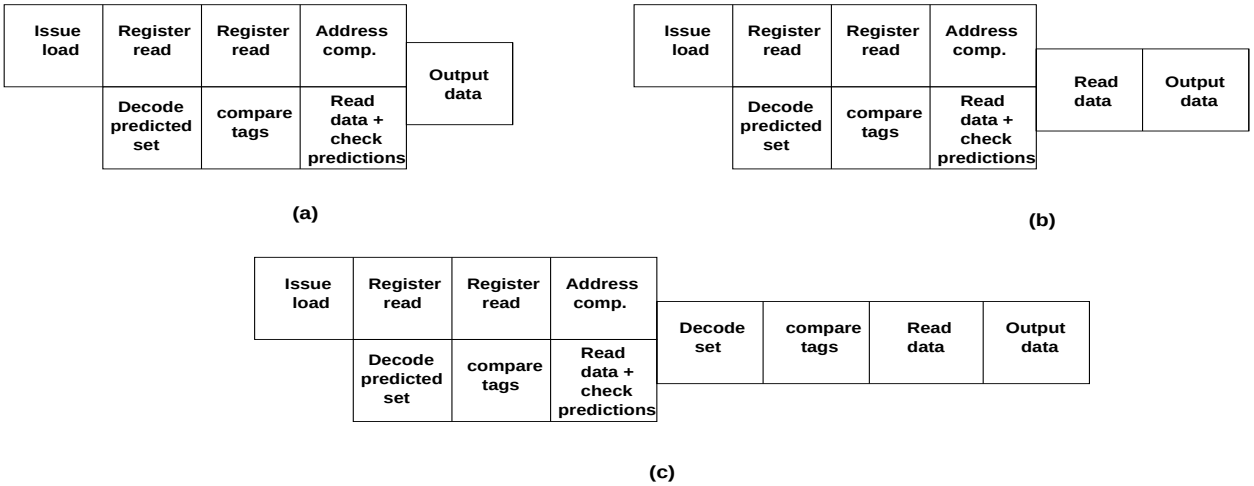

Figure 6. Pipelined Decoupled-Accessed Data Cache Access with Advanced Scheme for (a) Correct Set and Byte-Offset Predictions; (b) Correct Set Predictions but Byte-Offset Mispredictions; and (c) Set Mispredictions

decoupled-accessed cache using the *basic scheme* with 100% correct predictions, decoupled-accessed cache using the *advanced scheme*, and a decoupled-accessed cache using the *advanced scheme* with 100% correct predictions. Figure 7 shows the IPC results for these configurations as bars one to five, respectively. We also experimented with a 6-way set-associative 48KB L1 data cache with the same latency as the 32KB data cache (in Table 1); for the case where the hardware used for the 1K-entry ASAP table is used towards increasing the cache capacity. We found that, even though the miss rates changed for some of the benchmarks, the IPC obtained with a 32KB cache was almost equal to that of the 48KB cache. We do not present the IPC with the 48KB cache in Figure 7.

Figure 7 shows that there is a significant improvement in IPC for most of the benchmarks. For the FP benchmarks, that have a high prediction accuracies, the potential improvement in performance due to reduced cache access latencies is almost realized. However, for integer benchmarks, the full potential could not be realized because of their low prediction accuracies. Overall, the performance improvement in going from a 4-stage cache access pipeline to a cache access using the *advanced scheme* is about 8%, on an average, and reaches a maximum of 24% for `equake`.

Apart from the set predictor accuracy, the other parameters that affect performance when using set prediction are the number of load instructions in the program, the number of instructions dependent on the load instructions, the percentage of load instructions that get their values from store-load forwarding, and all the hardware parameters that affect the cache miss rate. With an increase in the number of load, the number of instructions dependent on the load instructions also increase, and more instructions can be benefited by schemes discussed in the paper, and better performance improvement can be expected. For instance, in Figure 7, benchmark `wupwise` has relatively lower number of load instructions (only about 10% of the total instructions) and hence relatively lower performance improvement from the schemes, even with 100% prediction accuracy. `Wupwise` also has almost 30% of load instructions getting their values from store-load forwarding. Same is the case with the benchmark `art`. With an increase in the percentage of load instructions receiving their values from store-load forwarding, lower percentage of instructions access the cache, reducing the performance improvements that can be gotten from the schemes. With an increase in the cache miss rate, more instructions need to go to the lower level cache for their data. Lower level caches have much higher latencies resulting in longer cache miss penalties.

Parameter	Value	Parameter	Value
Fetch/Commit Width	8 instructions	*Instr. Window Size*	96 Int/64 FP instructions
ROB Size	256 instructions	*Frontend Stages*	9 pipeline stages
Phy. Register File	128 Int/128 FP, 2-cycle pipelined access.	*Int. Functional units*	3 ALU, 1 Mul/Div, 2 AGUs
Issue Width	5 Int/ 3 FP	*FP Functional Units*	3 ALU, 1 Mul/Div
Branch Predictor	gshare 4K entries	*BTB Size*	4096 entries, 4-way assoc.
L1 - I-cache	8K, 2-way assoc., 2 cycle latency	*L1 - D-cache*	32K, 4-way assoc., 64 bytes block, 2 r/w ports 4 cycle pipelined access
Memory Latency	100 cycles first chunk 2 cycles/inter-chunk	*L2 - cache*	unified 512K, 8-way assoc., 10 cycles

Table 1. Default Parameters for the Experimental Evaluation

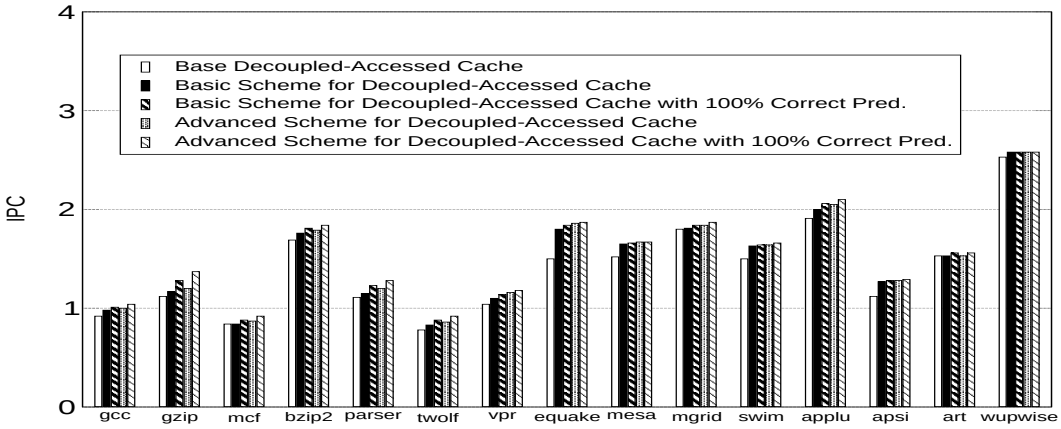

Figure 7. Performance (IPC)

With set prediction, (on a correct prediction) the cache miss penalty can be reduced by only a few cycles, and the small reduction in the penalty may not become visible as performance improvement. In addition, the lower level caches have smaller number of ports (because smaller number of accesses are expected) and may be implemented as non-pipelined and blocking caches. For instance, in Figure 7, benchmark `bzip2` has relatively higher cache miss rate, and even though it has a high percentage of load instructions, relatively high prediction accuracy, and relatively lower percentage of load instructions served through store-load forwarding, the potential improvement in performance is not very high. `Equake` has among the highest performance improvement, because it has a very high prediction accuracy, a large number of load instructions (almost 30%), very small cache miss rate of less than 1%, and relatively lower percentage (about 5%) of loads served through store-load forwarding.

6 Limiting Cache Energy Consumption

With our techniques, the number of accesses to the cache increases due to set mispredictions, which can increase the energy consumption in caches. In this section, we investigate techniques to reduce the additional energy consumption when using our schemes. We use the cacti tool [19], and a feature size of 0.18 μm for the energy measurements. In our cache organization, the cache has multiple banks (equal to the associativity), and each cache block (and its tag) in a set is stored in a separate bank. *Note that, read accesses consume optimum energy by just accessing the required data from a single cache line.*

6.1 Cache Energy Consumption

Figure 8 shows the normalized data cache energy consumption for the decoupled-accessed cache, the decoupled-accessed cache with the *basic scheme*, and the decoupled-accessed cache with the *advanced scheme*. The decoupled-accessed cache with the *basic scheme* consumes about 10% more energy than the baseline decoupled-accessed cache. The corresponding number for the decoupled-accessed cache with the *advanced scheme* is about 20%. The energy consumption for the *advanced scheme* is more because it incurs excess data array accesses on byte-offset mispredictions and also on set mispredictions. In the *advanced scheme*, when the tags in the predicted set are compared with the predicted tag, one of the tags in the set may match the address tag (even if the set is mispredicted), resulting in a data array access. In fact, our studies show a number of instances where a predicted address tag matched one of the tags in the mispredicted set. Figure 8 shows that the additional energy consumption is much more for the integer benchmarks as compared to the FP benchmarks, because of higher prediction accuracies for FP benchmarks. In fact, `gzip` shows the maximum misprediction rate and reads the maximum amount of excess data (in almost 40% of the data cache accesses), and hence, has the maximum cache energy consumption of almost 50% more than the decoupled-accessed cache. We observed that our predictor table access consumes half the energy of a data cache access. When factoring the predictor table energy consumption, the *basic scheme* consumes about 60% more energy, and the *advanced scheme* consumes about 70% more energy, than the baseline decoupled-accessed cache.

6.2 Reducing Mispredictions

The first technique that we investigate to reduce the number of additional accesses to the cache is to reduce the number of set mispredictions. Figure 3 showed that there are a significant number of set mispredictions for some of the benchmarks, and the idea is to convert these mispredictions into *no predictions*. To reduce the set

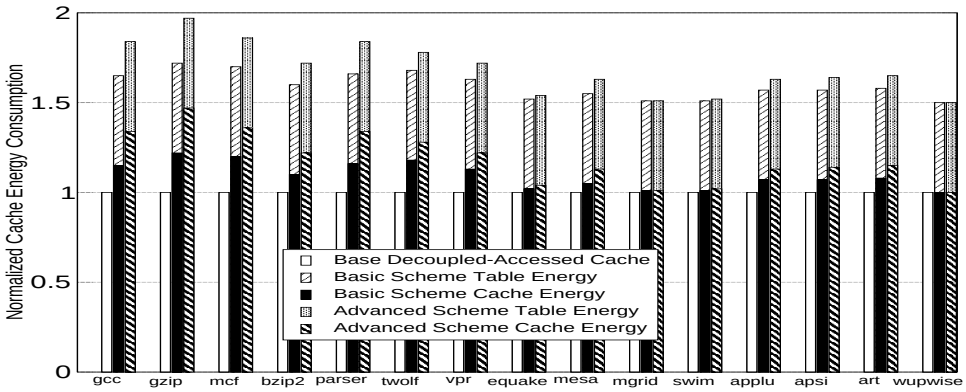

Figure 8. Normalized Data Cache and Predictor Table Energy Consumption

mispredictions, we use the confidence counters in the *ASAP* predictor. A prediction is made only if the counter value is such that it depicts sufficient confidence in the prediction. If the confidence in the prediction is not enough, then no prediction is made. In this technique, some of the correct predictions (in Figure 3) will also get converted into *no predictions* because of lack of confidence. A slight decrease in the correct prediction rate is expected, which can slightly decrease the performance obtained with set prediction. To reduce byte-offset mispredictions, byte-offset predictions are made only when the confidence counter for the address predictor (refer to Figure 4) is high. In our experiments, we use a confidence threshold of 2, that is prediction is made if the counter value is greater than or equal to 2. As discussed in Section 2, the address tag prediction is almost always correct when the set prediction is correct. Hence, we use the confidence-based prediction mechanism for only the set predictions in both the *basic* and the *advanced* schemes. We present the prediction accuracies when using confidence-based prediction, the IPC, and the energy results of this scheme along with the results of the next scheme.

6.3 Line Prediction

If a load instruction is accessing a data that has the same set and the same address tag as the previous access, then there is a very high probability that the cache block (which is accessed) is in the same position as the last time, unless it has been replaced. Hence, if the predicted set is the same as the last set in the predictor (the stride is 0 for a stride predictor) and the tag is predicted using the *last tag predictor*, and the confidence in the prediction for this particular load instruction is high, then the cache line (where the data is expected to be present) is predicted as the last line that was accessed by the instruction. For this, each entry in the set predictor is extended by 2 bits to also include the cache line accessed by the last committed instance. If the set is not the same as the last, then (even if it is highly predictable) no guarantee can be made regarding the position of the cache block in the new set. Hence, we predict the lines only for highly predictable load instructions that have the same predicted set and the predicted address tag as the last instance. In this scheme also, the set and byte-offset mispredictions are reduced using the confidence-based prediction of the previous section.

Figure 9 shows the predictability and the prediction accuracies for a *last line predictor (LLP)*. These measurements are made for all the data cache accesses, and not only for the ones that have the same predicted set as the last instance. Figure 9 shows that the cache line has a high predictability (higher for FP benchmarks as compared to integer benchmarks) based on the last line accessed. Comparing the last-value-based predictability of the sets (in Figure 2) the lines, it can be seen that the cache line number is repeater more than the set number. This is because, in case the set is the last set accessed, the line is also almost always the last line accessed (unless consecutive data cache accesses from the same load instruction are random accesses to the same set but different cache blocks), and if the sets are different, even then the current line accessed could still be in the same position in the new set as the last line was in the last set accessed. Figure 9 also shows that the prediction accuracies of line prediction is slightly lower than the predictability, mainly because of stale predictor data.

The cache access pipeline with line prediction remains the same as Figure 6. The only difference between correct line prediction and no line prediction is in the number of tag comparisons performed. With line prediction, only the tag from the predicted line is compared with the predicted tag. In case there is no line prediction, all the tags in the predicted set are compared against the predicted address tag. If the tag comparison fails in the predicted line, then it cannot be determined whether the line was mispredicted or the cache access has missed in the cache, unless all the lines in the set are checked. In our implementation, such a case is assumed to be a set misprediction. Our studies showed that this assumption does not hurt performance because there hardly are any cases where the set is correctly predicted but the line is mispredicted (because of the strict criterion used to predict the lines). In case of a set misprediction, the pipeline remains the same as in Figure 6(b).

6.4 Results

In this section, we present the prediction accuracies, the IPCs and the energy consumption results for the techniques discussed in Sections 6.2 and 6.3. Figure 10 shows the set prediction accuracies when using confidence-based prediction on an *ASAP* predictor, in comparison with the original set prediction accuracies. The figure also shows the line prediction accuracies with the strict prediction criterion. As can be seen in Figure 10, the mispredictions reduce significantly when using confidence-based prediction, and as expected, there is also a slight reduction in the percentage of correct predictions. Another important observation is that the percentage of mispredictions for line predictions is lower than that for the set predictions. This is because any set correct prediction results in correct line prediction (due to the criterion for line prediction used), and some of the set mispredictions also have correct line predictions. In addition, it can be observed that, for some of the benchmarks, the percentage of predictions made for line predictions is significantly less than that made for set predictions. This is true for benchmarks

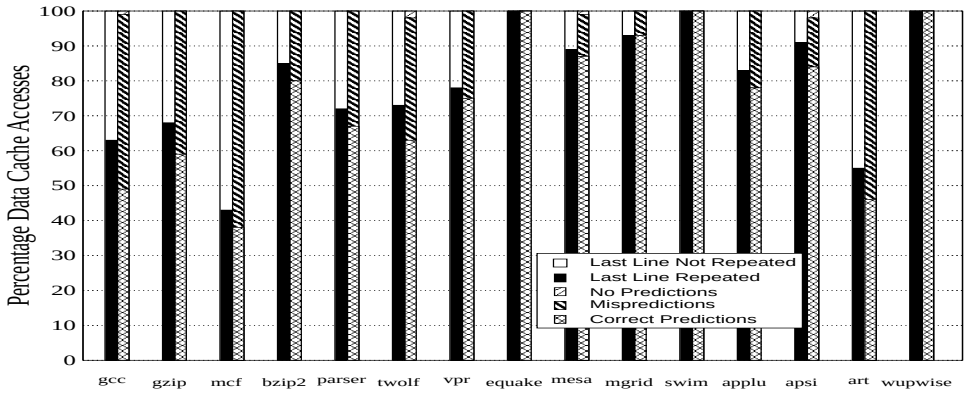

Figure 9. Predictability of Cache Lines (1st Bar) and Prediction Accuracies for Last Line Predictor (LLP) (2nd Bar)

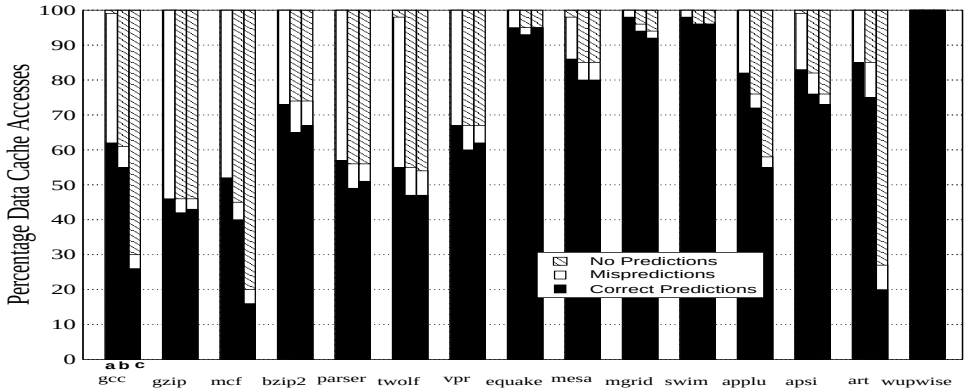

Figure 10. Prediction Accuracies for Set Predictions using *ASAP* (a) Without Confidence-based Prediction and (b) With Confidence-based Prediction; and Line Predictions (c) With Confidence-based Prediction

where a significant number of predictions is made using a stride-based predictor, in which case, the line predictions are not made, as the sets change.

Next, we look at the IPC results obtained when using confidence-based prediction to reduce the number of set mispredictions, shown in Figure 11. Figure 11 shows no change in IPC, as compared to the original *advanced scheme*, for almost all the benchmarks. This is because, the change in correct set predictions is small to make a significant difference, and because most of the IPC loss is recovered by a reduction in misprediction. Lower percentage of line predictions also does not impact performance, because, the pipeline is the same for no line predictions and correct line predictions.

Now, we look at the energy savings obtained from the energy reduction techniques discussed in this section. Figure 12 gives the normalized data cache energy consumption with respect to the base decoupled-accessed cache. Figures 12 and 8 show that the energy consumption for the advanced scheme reduces significantly with a reduction in the mispredictions, as expected. With just reducing the mispredictions, the data cache energy consumption of the *advanced scheme* is about 3% more than the baseline decoupled-accessed cache. However, when using *line prediction*, the data cache energy consumption is about 15% less than the baseline decoupled-accessed cache. When comparing to just reducing the mispredictions, *line predictions* further reduce the energy consumed in the tag arrays by accessing only the predicted line for tag comparisons. However, when the energy consumed by the predictor table is factored in, *advanced scheme* with *line prediction* consumes about 35% more energy than the decoupled-accessed cache.

7 Conclusion

With the CMOS scaling trends and slow scaling of wires as compared to the transistors, the cache access latencies will increase in the future microprocessors. To prevent the increasing latencies from affecting the cache throughput, the cache access is pipelined. However, even with a pipelined cache access, the instructions dependent on the load instructions still have to wait a significant number of cycles, which leads to significant performance degradation.

In this paper, we investigate predicting the set (that will be accessed by the cache access) and the tag of the address to effectively reduce the cache access pipeline length. First, we propose a new *adaptive set-address predictor* for predicting the sets (to be accessed) more accurately. In this predictor, either the set prediction is used or the set is extracted from the predicted address based on the confidence in each of the predictors. To reduce the effective cache access pipeline length, once the load instruction is issued, the cache access pipeline is started (using the predictions) in parallel to the address computation, thus reducing the load-to-use latency. We extend this scheme to also predict the byte offset of the data, so that the data is accessed even before the address is computed. Our studies showed that these techniques can improve the overall performance of the processor by as much as 24%.

We also propose techniques to reduce the additional energy consumed due to set mispredictions, by using confidence-based prediction technique to reduce the set mispredictions. We further reduce the energy consumption by predicting the exact cache block (in the set) where the data is expected to be present. With line prediction, the energy consumption of the techniques in the data cache is about

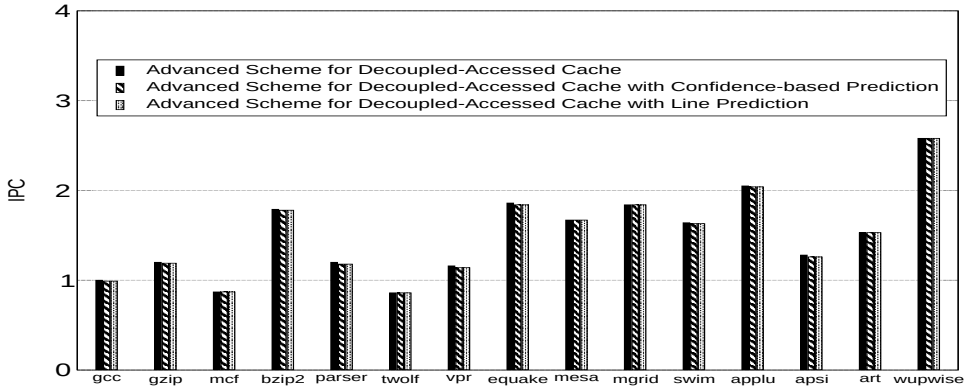

Figure 11. Performance (IPC)

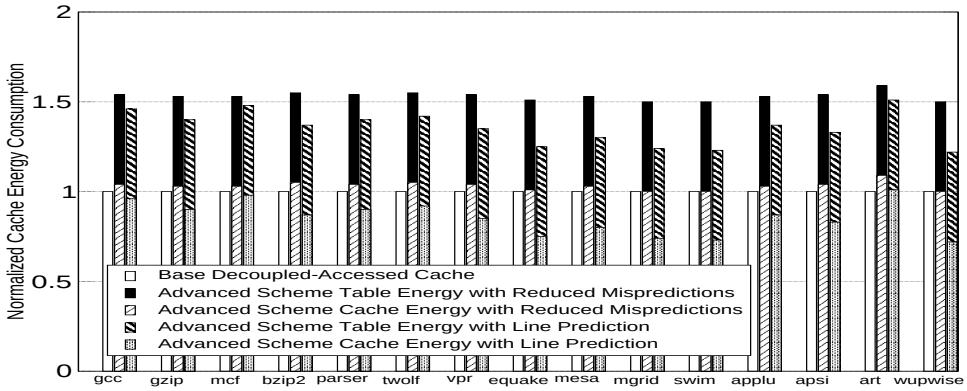

Figure 12. Normalized Data Cache and Predictor Table Energy Consumption

15% less than the baseline decoupled-accessed cache, while still maintaining the performance improvement.

8 Acknowledgments

The author would like to thank Dmitry Ponomarev, Prateek Pujara, and Sumeet Kumar for their reviews on the earlier version of the paper, and Matt Yourst for the earlier version of the simulator.

9 References

[1] T. Austin and G. Sohi, "Zero-cycle Loads: Microarchitectural Support for Reducing Load Latency," *Proc. Micro-28*, 1995.

[2] T. Austin et. al., "Streamlining Data Cache with Fast Address Calculation," *Proc. ISCA-22*, 1995.

[3] J. Baer and T. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. ICS*, 1991.

[4] M. Bekerman, et. al., "Correlated Load Address Predictors," *Proc. ISCA-26*, 1999.

[5] M. Bekerman, et. al., "Early Load Address Resolution Via Register Tracking," *Proc. ISCA-27*, 2000.

[6] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*, June 1997.

[7] B. Calder, and D. Grunwald, "Next cache line and set prediction," *Proc. ISCA*, 1995.

[8] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," *Proc. HPCA-2*, 1996.

[9] T. Chappell, et. al., "A 2-ns cycle, 3.8-ns access 512-kB CMOS ECL SRAM with a fully pipelined architecture," *IEEE Journal of Solid State Circuits*, 26(11):1577-1585, 1991.

[10] T. Chen and J. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers*, V.45, N.5, May 1995.

[11] Z. Chishti, and T. N. Vijaykumar, "Wire Delay is not a problem for SMT (in the near future)," *Proc. ISCA-31*, 2004.

[12] J. Gonzalez, and A. Gonzalez, "Speculative execution via address prediction and data prefetching," *Proc. ICS*, 1997.

[13] M. K. Gowan, et. al., "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. DAC*, 1998.

[14] G. Hinton, et al, "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, Nov. 2001.

[15] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, March/April 1999.

[16] M. H. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction," *Proc. ASPLOS*, 1996.

[17] S. Manne, A. Klauser and D. Grunwald, "Pipeline Gating: Speculation Control For Energy Reduction," *Proc. ISCA*, 1998.

[18] M. Powell, et. al., "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," *Proc. Micro-34*, 2001.

[19] P. Shivakumar, and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing Power, and Area Model," *Technical Report, DEC Western Research Lab*, 2002.

[20] V. Tiwari, et al., "Reducing Power in High-performance Microprocessors," *Proc. DAC*, 1998.