Dynamic Partition of Memory Reference Instructions – A Register Guided Approach[*]

Yixin Shi and Gyungho Lee

ECE Department, University of Illinois at Chicago
yshi7@uic.edu, ghlee@ece.uic.edu

Abstract. A high bandwidth L-1 data cache is essential for achieving high performance in wide-issue processors. Previous studies have shown that using multiple small single-ported caches instead of a monolithic large multi-ported one for L-1 data cache can be a scalable and inexpensive way to provide higher bandwidth. Many schemes have been proposed on how to direct the memory references to these multiple caches in order to achieve a close match to the performance of an ideal multi-ported cache. However, most previous designs seldom take dynamic data access patterns into consideration and thus suffer from access conflicts within one cache and unbalanced loads between the caches. We observe that if one can group data references defined in a program into several regions (access regions) to allow parallel accesses, then providing separate small caches (access region cache) for these regions may prove to have better performance than previous multi-cache schemes. The register-guided memory reference partition approach proposed in this paper effectively identifies these semantic regions and organizes them in multiple caches in an adaptive way to maximize concurrent accesses without incurring too much overhead. In our design, the base register number, not its content, in the memory reference instruction is used as a basic guide for instruction steering. A reassignment mechanism is applied to capture the pattern when program is moving across its access regions. In addition, a distribution mechanism is introduced to further reduce residual conflicts, which adaptively enables access regions to extend or shrink among the physical caches. Our simulations of SPEC CPU2000 benchmarks have shown that the register-guided approach can reduce the conflicts effectively, distribute memory reference instructions properly, and yield considerable performance improvement in terms of IPC.

1 Introduction

Modern superscalar processors select and execute multiple independent instructions at a very high clock rate assisted by control speculation, register renaming, and data-flow execution. With ample on-chip hardware resources available, researchers have been actively proposing aggressive micro-architectures that can issue more instructions including memory reference instructions in a single clock cycle[3]. Traditional efforts were mainly focused on decreasing the cache access latency and increasing the cache capacity. However, previous studies [4][11] suggest that the capability to provide enough memory bandwidth (or cache ports) be also important to explore more instruction level parallelism[9].

[*] This work was supported in part by NSF CCR0225561

Essentially the ways to achieve high memory bandwidth can be categorized into three classes. The most straightforward approach is to build an ideal multi-ported cache. This circuitry level solution often comes at the cost of complexity in memory cell and bit/word line design and possibly incurs longer cache access latency[7]. Fig.1 shows the various performance trends of a 32 KB cache modeled by CACTI 3.0 in .18um[13]. The three metrics, access time, cache area and the power consumption, increase quickly as more cache ports are introduced. Alternatively, there have been many proposals to approximate the ideal multi-ported cache including time-division multiplexing and cache replicating. These designs often suffer from either poor resource utilization or longer access latency.

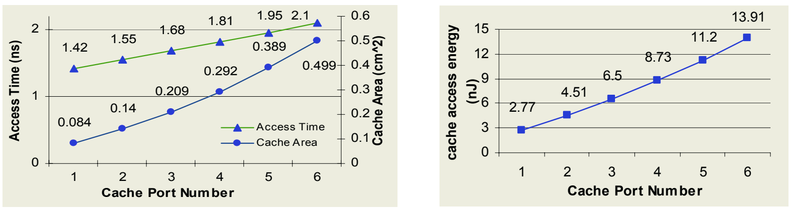

Fig. 1. Access time, area, and power consumption of a 32 KB, 32B block size, 2-way set-associative cache with different cache port number[5]

The interleaved multi-banking scheme is another way to increase the memory bandwidth with less hardware complexity. Instead of using one big ideal multi-ported cache, multiple smaller banks or caches serve as L-1 data cache. The data are simply interleaved based on word addresses or cache lines so that two or more simultaneous accesses to different banks can be supported in one clock cycle. This design typically employs an interconnection network (crossbar) to distribute memory references among the different cache banks (see Fig.3). One problem is the bank conflicts among the simultaneous accesses caused by the "random interleaving" property. Another potential problem is that the area of a crossbar in the critical path increases super-linearly when the number of banks increases. This will cause further delay when accesses are passing through the crossbar.

Other than the multi-banking solution, many schemes have been proposed to increase the bandwidth in a more scalable manner. Similar to multi-banking, multiple caches are used as L-1 data cache. However, these designs use more intelligent methods in data placement and memory reference steering rather than simply interleaving the addresses. The proposed register-guided memory partition scheme belongs to this category. It tries to exploit the semantic meaning in the program when assigning memory instructions to different caches. The key insight is that the *base register number,* not its content, can serve as the basis for instruction steering, because the register number usually reflects the data "region" on which the instruction is operating. By adaptively interpreting different registers for different regions, the data regions can be distinguished from each other and memory access parallelism can be captured from the program level. In addition, a reassignment mechanism and a distribution mechanism are applied to capture the changes in the memory reference pattern and alleviate the conflicts. Simulations show this scheme outperforms other solutions for most benchmark programs.

The remainder of this paper is organized as follows: Section 2 summarizes related works on multi-cache design; Section 3 discusses the details about the register-guided memory instructions partition scheme; Section 4 describes the scheme-specific architectural parameters, the simulation approaches, and the benchmarks used; Section 5 presents our experimental results and analysis; Section 6 provides the concluding remarks.

2 Related Work

Sohi and Franklin [5] first predicted that the L1 cache bandwidth would eventually become a performance bottleneck for a multiple-issue processor. Wilson [19] also argued that adding more ports to the L1 cache could become costly and inefficient in terms of area and access time. Neefs [10] reported potential benefit of bank prediction to remove the crossbar from critical path. Yoaz [20] also proposed bank prediction that increased the cache port utilization through a balanced scheduling of loads toward multiple cache banks. The data-decoupled architecture (DDA) proposed by Cho [4][5] splits the data cache according to the program space types (i.e. stack, heap, and data). It simply treats each area as an access region and divides the data references into two independent streams (stack and non-stack). Thakar [17] tries to further split data cache within stack cache and non-stack cache. This scheme assigns the access regions to the access region caches initially based on offline profiling and then predicted by a PC-indexed table. Redirection is used to maintain the data consistence and only one copy for a datum is allowed in the L-1 cache. The Parallel Cachelets scheme proposed by Limaye[8] also employs a PC-indexed table to determine the bank (or cachelet) number either in decode stage or execution stage. It tries to minimize contentions by reassigning the destination for memory access once a conflict occurs. To maintain consistency, a write through policy and value broadcasting are used. Racunas [11] also studied the performance impact on a partitioned L-1 data cache. They proposed a two-bit saturating instruction hysteresis counter in the prediction table to partition memory reference streams.

3 Register-Guided Memory Partition with Distribution Scheme

3.1 Motivation

The register-guided memory partition scheme is based on the concepts of Access Region and Access Region Cache first proposed in [4][17]. A key observation is that typically, there exist one or more data structures with variable sizes in a program either statically defined or generated at run-time. They can be data arrays found in FORTRAN programs or structures/unions or objects common in C/C++ programs. These data structures are called *access regions*. Our partitioning scheme tries to capture these semantically defined and logically independent access regions as atomic units in memory. Ideally, by navigating the partitioned memory reference stream, data from the different access regions are placed into physically separate caches. These multiple quasi-independent small caches working as L-1 cache are named *Access Region Caches (ARC)*[17].

We extend our previous work [4][16] by proposing a novel and more effective method to predict the destination access region cache for each memory access. Unlike some "software" solutions such as load instruction annotation [18] or static marking by compiler, our architectural level approach tries to utilize run-time information without changing the existing binaries. After investigating the prediction resources (e.g. program counter, previous branch history behavior, register number, and probably the content and offset) and their available time, we found that the base register number in the memory reference instruction can serve as a good hint.

In a typical MIPS-like architecture, the memory reference instructions, i.e. load and store, generally have the following format:

```
LOAD        destination-register, offset(base-register)
STORE       source-register,      offset(base-register)
```

Compiler typically groups the data members belonging to one data structure by assigning a common register as their base registers. Memory reference instructions then use this register together with variable offsets to access different fields within that data structure. We also expect the memory reference instructions accessing different data regions in a short time window to have different base registers. Therefore, the base registers reflect the access regions and can be utilized to identify the data structures in the program. The partitioning based on the base register is motivated by the fact that simultaneous accesses on different data structure are usually relatively independent and can be performed concurrently. This approach ideally provides the separate spaces for the access regions that may have different access patterns. This explores opportunities to improve performance similar to separate instruction cache and data cache found in most processors today. Although some data regions might have to share one ARC due to the limited number of physical ARCs, our round-robin ARC assignment and reassignment mechanisms to be presented later can minimize this effect. Using register number to determine the ARC number in this scheme is the major difference from previously proposed schemes. Using the base register number, we can capture more program semantic meaning than just blindly using the PC or addresses. In addition, the register number is known in an early pipeline stage so that after partitioning dedicated and small hardware structures can be used to process these instructions efficiently in later pipeline stages[1].

3.2 Proposed Scheme

3.2.1 Scheme Framework

Fig.2 shows the framework of our Register Guided memory partition with Distribution scheme *(RGD).* A register-indexed prediction table, called ARC prediction table (*ARCP*) is deployed to predict the ARC numbers in the fetch stage for memory instructions. Therefore, no crossbar is needed. The instructions are steered into multiple pipelines and Load/Store (L/S) units. Each entry in ARCP table is mapped to a random ARC cache initially and will be trained at run-time later by the prediction updating policy. The verification logic, which is activated when the effective address is known, resides in the Load/Store unit. If the ARC number is correctly predicted, the instruction goes to the cache and performs an access. Otherwise, a redirection network is used to redirect the instruction to the correct ARC with some cycles of redirection penalties. We assume a select and re-issue mechanism is employed on misprediction. Some run-time information, such as conflicts and redirection events (will be described later), is fed back from the Load/Store unit to the prediction unit to update the prediction table and adjust the steering policy.

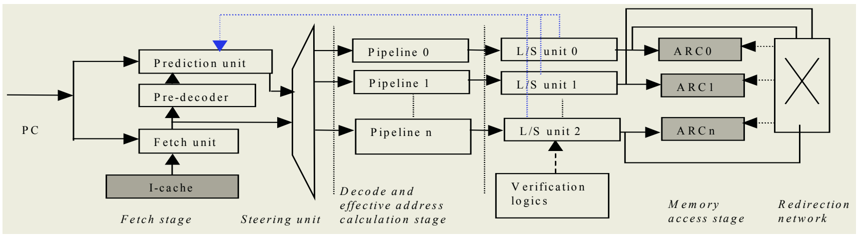

Fig. 2. The frame work of the proposed RGD scheme

We also show a typical cache-interleaving (multi-banking) scheme in Fig.3 for comparison purpose. In this scheme, the cache bank is determined *after* the effective address is calculated. Then the memory reference instruction is steered into the bank through the crossbar. Consequently the crossbar is in the critical path here while the redirection network in RGD scheme is not, provided that the ARC prediction accuracy is reasonably high.

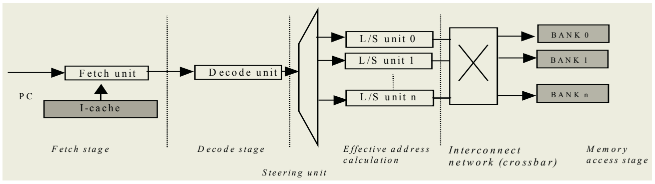

Fig. 3. Cache bank-interleaving scheme

3.2.2 Prediction Verification

In RGD scheme, every memory access must be verified against the correct access region information when the actual address is produced. The effective address is calculated during the first step of the memory-access stage. Meanwhile, access region verification is completed by comparing the tags in the cache or in a separate tag table. Unlike other schemes such as parallel cachelet[8], the RDG scheme does not allow multiple copies of a datum to exist in L-1 caches. Therefore, if the tag comparison turns out to be a mismatch, the verification unit checks other caches. This can be done by broadcasting current datum's tag to other ARCs using a bus or by maintaining a "super" tag, i.e. aggregate of all the ARC tags, in a way similar to duplicate tagging for multiple cache coherencies. If the checking results mismatch on the rest of ARCs either, a true cache miss occurs and L-2 cache access is then invoked. If the datum is found in another ARC, the instruction is redirected and reinserted into the correct memory pipeline connecting to that ARC through a redirection network as shown in Fig.2. We call such an event as ARC misprediction. In this study, as a select and re-issued approach is used, the effects of mispredictions are evaluated by imposing a penalty of a certain number of clock cycle delays for that instruction.

3.2.3 Prediction Updating

In the context of prediction on memory references, last value predictor and 2-bit saturated predictor have been studied in literature[8][11][17]. In our design, a threshold-triggered updating method is used to provide a kind of hysteresis effect to smooth the transient deviations. Rather than update the prediction table immediately when a misprediction is detected as in[8], we periodically check some interested events (including mispredictions) that are accumulated in counters during a sampling period. If any counter exceeds a pre-defined threshold, prediction updating is triggered. Following two mechanisms are implemented as updating policies.

Reassignment Mechanism: The reassignment mechanism can be used in two scenarios to improve the prediction accuracy, as shown in Fig.4. One register in a program can be utilized as the base register for different data regions at various stages of execution. This changing may cause cache misses and ARC misses (redirection events), which implies that the interested register may have been reused or spilled and it may now represent a new data region. To capture this change, a threshold, Rt (Reassignment threshold) is established for updating the ARCP table on ARC mispredictions. That is, the entry for a register in the ARCP table is reassigned to a new destination of ARC only after more than Rt redirection events have been detected in a sampling period as shown in Fig.4(a). By choosing a proper value for Rt, we can capture the moving behavior and adaptively adjust the prediction value.

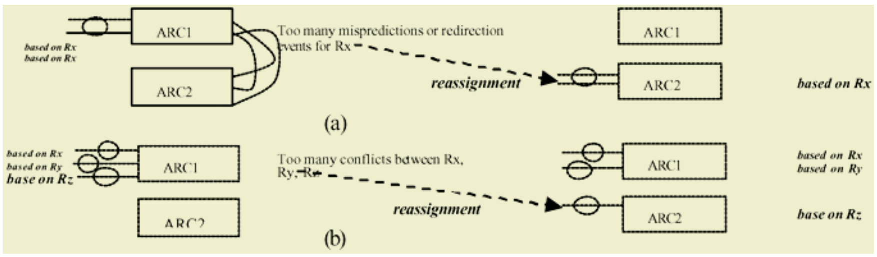

Fig. 4. Two scenarios when the reassignment mechanism is invoked

The reassignment mechanism can be also applied to reduce ARC conflicts. Similar to bank conflicts, ARC conflicts occur when two or more data regions are assigned into one physical ARC and the program happens to access these regions simultaneously as shown in Fig.4(b). In this case, one of these regions needs to migrate to another ARC to reduce the conflicts. Again, a conflict counter and a predefined threshold Ct (Conflict Threshold) are used to determine whether to update the prediction table. We direct the memory accesses of a region to the one that has the least conflicts observed. This mechanism forces one access region to leave its current ARC to avoid further conflicts.

Distribution Mechanism: We also observed that programs might reference one access region based on a *same register* intensively. For instance, a program is likely to make intensive operations on its local variables during a function call where the memory reference instructions have stack pointer or frame pointer as their base registers. In this case, the redirection mechanism will not help because all the instructions with the same base register are driven into the same ARC. To handle this, we introduce a distribution mechanism to scatter these accesses. First conflicts are classified into two types. The conflicts caused by the instructions with the same base register are named as self-conflicts and all other conflicts as interference-conflicts. The ratio of the self-conflicts over all conflicts for each base register is monitored for each register. When this ratio for one particular register reaches a pre-defined threshold, the program is identified as operating on one data region and the distribution flag is set for that entry in ARCP. The memory reference instructions based on the register are then distributed to all of the ARCs in a round-robin manner.

Two counters are employed to accumulate the number of the two types of conflicts. A parameter SIt (Self-conflicts & Interference-conflicts threshold) is used to represent the distribution threshold. Rather than calculating the ratio, the following condition, *Self-conflict number - Interference-conflict number > SIt,* is checked periodically to determine if we should distribute one data region in our simulation.

3.2.4 Hardware Cost

The hardware cost for implementing the RGD scheme is moderate. It basically consists of four counters, a small ARCP table, and some lookup and control logic. In our simulation, each entry in ARCP contains 10 fields (each of one byte). Assuming up to 32 registers can be used as base registers, the size of the ARCP table is only 32 x 10 = 320 bytes with some glue logic. In other PC-based prediction schemes, however, a modest prediction table would have 2K-4K entries totaling 10KB. Hence, the speed of accessing and updating the ARCP table in RGD scheme can be much faster. Furthermore, a smaller ARCP table is generally preferred because the ARCP table itself should be ideally multi-ported to support multiple lookups in a single clock cycle. This fact is largely ignored in most previous PC-indexed schemes.

4 Simulation Methodology and Architectural Parameters

4.1 Simulation Parameters and Scheme-Specific Architectural Parameters

In our simulation, a cycle-accurate execution driven simulator derived from the Simplescalar Tool Set 3.0[2] is modified to incorporate our design of multiple memory pipelines and ARCs. To evaluate our proposed approach as emerging trend towards aggressive ILP exploitation, an out-of-order processor model issuing up to 16 instructions per cycle is used. An ideal front-end for the processor model is assumed in order to assert a maximum data bandwidth demand on the memory system.

The L-1 Data cache are direct-mapped caches with a fixed total size of 64KB across all of the different ARC configuration and memory partitioning schemes. In order to investigate the scalability, we studied the cases of 4-ARC and 8-ARC configurations. For the 4-ARC configuration, four separate single-ported caches (ARCs) are used as the L-1 Data cache, each of 16KB; while in the 8-ARC configuration, eight ARCs are provided, each of 8KB. All caches are assumed to be lock-up free. We tested the pre-compiled Alpha binaries of both integer and floating-point benchmarks from SPECCPU2000[15] benchmark suite with *reference* inputs. To warm up the architecture, we fast-forwarded the first 500 million instructions and collected data for the next 500 million committed instructions. The parameters we assumed are summarized in Table-1.

Table 1. Architectural Parameters in our simulation model

Fetch/decode/issue/ commit width	16
Function unit size	Int ALU:16, FP ALU: 16, Int Mult: 4, FP Mult: 4
L1 I-cache	Blk size:32B; set: 512; assoc:2; access time:1 cycle;
L1 D-cache	Blk size: 32B, set: 512(4ARC), 256(8ARC); per ARC size: 16KB(4ARC), 8KB(8ARC); Total size: 64KB, access time 2 cycle;
Unified L-2 cache	Blk size:64B, set: 2048; assoc.: 4; total size: 512KB. access time: 8 cycles;
Others	Perfect branch predictor; LSQ size: 128; RUU size: 256; memory latency: 50 cycles;

Table-2 shows the scheme-specific architectural parameters in the simulation. Here, the event counters are checked when every ten memory reference instructions have been committed (SP=10). This corresponds to approximately three basic blocks. If redirection events occur roughly half the time, then reassignment is triggered (Rt = 5). Similarly, five or more conflicts also lead to migration of a data region to another ARC (Ct=5). The value for SIt is assumed to be three to determine whether to trigger distribution mechanism. These parameters, currently having fixed values, are expected to be tunable responding to different applications at run-time in the future.

Table 2. Scheme-specific Architectural Parameters in the simulation model

Parameter Name	**Value**	**Parameter Name**	**Value**
Sampling Period (Sp)	10	Self conflicts & Interference conflicts Threshold (SIt)	3
Redirection Threshold (Rt)	5	ARC/L-1 cache hit Time	2 cycles
Conflict Threshold (Ct)	5	Redirection Penalty	2 cycles

4.2 Schemes for Comparison

The baseline model in this study is the multi-banking schemes where data are placed in an interleave manner and the memory reference instruction is steered through a crossbar. One baseline model is the BI-2 scheme (Bank Interleaving) where 2 cycles are charged for the crossbar delay, the same as the redirection penalty in RGD scheme (see Table-2). Another one is a more aggressive multi-banking scheme, the BI-1 scheme, which charges only 1 cycle for the crossbar delay. The third scheme, the PC prediction (PCP), similar to the Parallel Cachelets[8] and Tharker's[17] design, is a general PC-based prediction scheme. It accommodates a 2KB prediction table indexed by the PC to predict the destinations for memory reference instructions. Redirection mechanism with a penalty of 2 cycles is used to maintain data consistency. A fourth scheme, called the register-guided scheme (RG), is also simulated to understand how much the distribution mechanism in RGD scheme contributes to the final performance. It is similar to the RGD scheme except that no distribution mechanism is applied. Note that the same size L-1 data caches (64KB) are used in the above four schemes as that of ARCs in RGD scheme.

5 Simulation Result and Analysis

5.1 Busy-Waiting Cycle

Fig.5 shows the busy-waiting cycles for memory reference instructions for the 4-ARC and the 8-ARC configuration. They are defined as the latencies between the time when the operands of a load or store instruction are available to the time when this instruction gets an idle port. The busy-waiting cycles include the waiting time in LSQ, redirection penalty, and the crossbar delays. It reflects the degree of bank conflicts and how well memory ports are utilized. As can been seen in Fig.5, for 4ARC-integer benchmarks, the average busy-waiting time for RGD is 0.6 to 1.5 cycles fewer than other schemes, which mainly contributes to a higher IPC. Similar results can be observed for 4ARC-INT and 8ARC-FP benchmarks. For FP programs in 8-ARC configuration, the busy-waiting cycle of RGD scheme is on average lower by about 0.5 cycle than that of BI-2, but 0.35 cycles higher than BI-1. This indicates in this case the conflict reduction by RGD scheme is not sufficient to beat the benefit obtained from a shorter crossbar delay (one cycle) we assumed in BI-1.

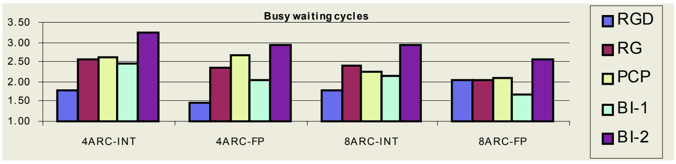

Fig. 5. Average Busy waiting cycles

5.2 ARC Prediction Accuracy and Data Cache Hit Rate

Fig.6(a) presents the ARC prediction accuracy. The RGD, RG, and PCP have similar ARC prediction accuracy of 81%, 82%, and 83.7%, respectively. Considering PCP scheme has much bigger PC-indexed prediction table, the register-guided prediction is a fair tradeoff in efficiency and accuracy. In addition, with an 81% ARC prediction accuracy on average, we can also conclude that the redirection network shown in Fig.2 is not in the critical path.

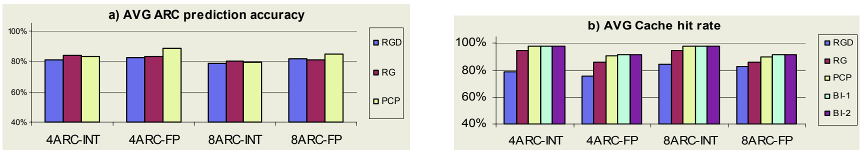

Fig. 6. Average ARC prediction accuracy and L-1 Data Cache hit rate

The overall data cache hit rate for the ARC is illustrated in Fig.6(b). The RGD scheme has about 10%-14% lower cache hit rate than that of RG, PCP, and BI scheme. This is due to the redirection and distribution mechanisms incurring considerable invalidations and thus causing extra cache misses while reducing the total number of conflicts. Note that a higher cache hit rate here does not necessarily mean higher performance, because memory reference instructions would experience redirection and conflict penalties before the final cache access occurs.

5.3 Overall IPC

Fig.7 shows the overall IPC for all of the schemes discussed so far. The simulation results indicate that with the same size of the L-1 cache and the same redirection penalty, our scheme works best for most of the benchmark programs under different ARC configurations. For the integer benchmarks in the 4-ARC configuration in Fig.7a, many benchmarks in RGD have considerable IPC improvements, 9%, 18%, 8%, and 35% over RG, PCP, BI-1, and BI-2, respectively. The results also indicate that the conflict reduction by reassignment and distribution mechanisms does compensate for the lower cache hit rate incurred for most benchmarks. In this configuration, the only two exceptions are *perlmk* and *twolf*. Similar results of performance improvement are obtained for the FP benchmarks in the 4ARC configuration in Fig.7b and integer benchmarks in the 8ARC configuration in Fig.7c. The result for FP in the 8ARC configuration is not so impressive in Fig.7d where the IPC of the RGD is nearly the same as that in RG and PCP schemes. It is worse (-3.1%) than that in BI-1 scheme. This is probably due to the fact that architectural level solutions have a smaller gain with fairly regular access patterns in FP programs and RGD scheme cannot capture more parallelism to cover the reduced cache hit rate.

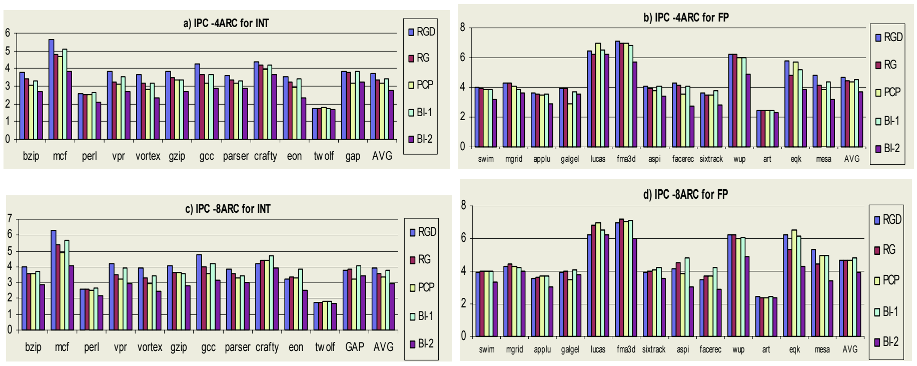

Fig. 7. Overall IPC

We can also observe that the RG scheme outperforms PCP and BI-2 in most cases. RGD having a further 6% higher IPC than that for RG on average implies that the distribution mechanism does reduce the total number of conflicts and attain an overall gain. Moreover, we can see that the IPC from both the RG and PCP schemes are slightly lower than the aggressive bank interleaving scheme (BI-1) while IPC for the RGD scheme is higher in most cases. This suggests that combining the register-guided partitioning and a prediction updating policy with reassignment and distribution mechanisms makes RGD scheme effective.

6 Conclusions

This paper proposes a register-guided memory reference partitioning approach by taking the dynamic behavior of memory references into consideration. We first observe that there are relatively independent groups of data structures in the program, called "access regions" in this paper. Parallel accesses for higher bandwidth can be achieved if these access regions are identified at run-time. We also explore a notion that the base register in memory reference instructions can be a guide to track these regions. By taking into account the base register information for memory reference instruction for predicting and steering, the register-guide dynamic memory partition scheme demonstrates the ability to adaptively trace the individual access regions. The threshold-based reassignment and distribution mechanisms are employed to track the changing of access region the base registers represent and alleviate conflicts at run-time. The simulation shows that this register-guided (RGD) scheme outperforms other existing schemes in most benchmark programs. Therefore, we consider it a promising technique to support high bandwidth memory accesses with a good scalability.

References

1. V.Agarwal, M.Hrishikesh, S.Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures", ISCA-27, May 2000.
2. T.M.Austin and D.Burger, "The SimpleScalar Tool Set," Univ. of Wisconsin Computer Science Dept. Technical Report, No. 1342, June 1997.
3. T.M.Austin and D.Burger, "Billion Transistor Architectures," IEEE Computer, Vol.30, No 9, June 1997.
4. S.Cho, P.C.Yew and G.Lee, "Access Region Locality for High-bandwidth Processor memory System Design," Proceedings of 32nd Int'l Symposium on Microarchitecture, November 1999.
5. S.Cho, "A High-bandwidth Memory Pipeline for Wide Issue Processors", University of Minnesota Computer Science and Engineering Dept. Ph.D. Thesis, Dec. 2002
6. A.Gonzalez, M.Valero, N.Topham and J.M.Parcerisa, "Eliminating Cache Conflict Misses through XOR-Based Placement Functions", Proceedings of the 1997 Int'l Conference on Supercomputing, July 1997.
7. IDT. Introduction to Multi-Port Memories, Application Note AN-253, 2000.
8. D.Limaye, R.Rakvic and J.P.Shen, "Parallel Cachelets," 2001 Int'l Conference on Computer Design, September 2001.
9. M.H. Lipasti and J.P. Shen, "Supperspeculative Microarchitecture for Beyond AD 2000," IEEE Computer, Sept. 1997
10. H.Neefs, H.Vandierendonck, K.de Bosschere, "A Technique for High-bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks," Int'l Symposium on High-Performance Computer Architecture, January 2000.
11. P. Racunas, Y. Patt, "Partitioned first-level cache design for clustered microarchitectures" Proceedings of the 26th Annual International Conference on Supercomputing, June 2003.
12. J.A.Rivers, G.S.Tyson, E.S.Davidson, T.M.Austin, "On High-Bandwidth Data Cache Design for Multi-issue Processors", Proceedings of Micro-30, December 1997.
13. P. Shivakumar and N.P.Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," COMPAQ WRL Research Report 2001/2, August 2000.
14. G.S.Sohi, M.Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors", ASPLOS-IV, April 1991.
15. SPEC2000, The tandard Performance Evaluation Corporation, http://www.specbench.org.
16. B.S.Thakar, G.Lee, "Access Region Cache: A Multi-porting Solution for Future Wide-Issue Processors", Proceedings of 2001 Int'l Conference on Computer Design, Sept. 2001.
17. B.S.Thakar, S.K. Park and G. Lee, "A scalable multi-porting solution for future wide-issue processors," Microprocessors and Microsystems, 2003.
18. Z. Wang, D. Burger, K.S.McKinley, and C. C. Weems, "Guided Region Prefetch: A Cooperative hardware/Software Approach", Proceedings of 30th ISCA, June 2003.
19. K.M.Wilson, K.Olukotun, M.Rosenblum, "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors", Proceedings of 23th ISCA, May 1996.
20. A.Yoaz, E.Mattan, R.Ronen, S.Jourden, "Speculation Techniques for improving Load Related Instruction Scheduling", Proceedings of 26th ISCA, May 1999.