1

ELSEVIER
Available online at www.sciencedirect.com

Microprocessors and Microsystems xxx (2006) xxx–xxx
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

2
Clustered indexing for branch predictors

3 Veerle Desmet *, Hans Vandierendonck, Koen De Bosschere

4 *Ghent University – UGent Department of Electronics and Information Systems (ELIS), Parallel Information Systems (PARIS) Group, Member*
5 *HiPEAC Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium*

6

7 **Abstract**

8 As a result of resource limitations, state in branch predictors is frequently shared between uncorrelated branches. This interference
9 can significantly limit prediction accuracy. In current predictor designs, the branches sharing prediction information are determined
10 by their branch addresses and thus branch groups are arbitrarily chosen during compilation. This feasibility study explores a more ana-
11 lytic and systematic approach to classify branches into *clusters* with similar behavioral characteristics. We present several ways to incor-
12 porate this cluster information as an additional information source in branch predictors.
13 Our profile-based results demonstrate that cluster information is useful in various branch prediction schemes. When *clustered indexing*
14 is applied, the same performance can be obtained with 2–8 times less hardware budget. For small predictor budgets, clustered indexing is
15 very cost-effective, e.g., the misprediction rate in an 8 Kib gshare is reduced 12.3% on average for SPEC CPU2000 INT. For large bud-
16 gets up to 4 Mib, clustered indexing reduces the number of mispredictions by 3–5%, or stated otherwise only half the hardware budget is
17 required to obtain the same performance as the original gshare scheme.
18 © 2006 Published by Elsevier B.V.

19 *Keywords:* Branch prediction; Aliasing; Clustering
20

UNCORRECTED PROOF

21 ## 1. Introduction

22 Past research has emphasized the aliasing problem in
23 dynamic branch predictors and has pointed at the mostly
24 destructive nature of this interference of branches [18].
25 We measured that the misprediction rate in a 8 Kib gshare
26 predictor would be reduced by 34% if all destructive alias-
27 ing could be avoided.
28 Today, the pairs of branches that cause aliasing are
29 determined by their branch addresses. As these addresses
30 are arbitrarily chosen by the compiler, this results in a var-
31 iable and uncontrolled amount of aliasing. The key idea
32 behind this research is to fix the pairs of branches sharing
33 prediction information in a systematic way. Therefore, a
34 cluster technique is used to form branch clusters based
35 on their behavioral characteristics. The paper focuses on
profile-based techniques for systematically selecting clus- 36
ters of branches that can share prediction information. 37
Therefore, we applied a cluster algorithm in which the time 38
varying taken rate behavior of branches serves as similarity 39
criterion during cluster formation. Kim and Tyson [8] 40
already suggested to reuse prediction resources only for 41
branches that interleave in time. In addition, our clustered 42
indexing strategy allows static branches with similar 43
dynamic behavior to share resources. In other words, we 44
additionally convert destructive interference into neutral 45
interference, providing an even more effective predictor 46
solution. 47
Further, this study explores the feasibility to guide var- 48
ious dynamic branch predictors according to this static 49
branch cluster information. The results show that clustered 50
indexing is extremely useful in today's range of practical 51
branch prediction implementations. Typically, those imple- 52
mented branch predictors are simple and small, often based 53
on bimodal, global and gshare components. Although we 54
focus on branch prediction, clustered indexing is a general 55
technique that can be used in various other hardware struc- 56

* Corresponding author. Tel.: +32 9 264 3594; fax: +32 9 264 3405.
E-mail addresses: veerle.desmet@elis.UGent.be (V. Desmet), hans.vandierendonck@elis.UGent.be (H. Vandierendonck), koen.debosschere@elis.UGent.be (K.D. Bosschere).

0141-9331/$ - see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.micpro.2006.08.003

tures in processors, e.g., branch target buffers, trace caches, pre-fetchers, value predictors, etc.

The remainder of this paper is organized as follows. Section 2 starts with the basic concept of our alternative indexing technique and Section 3 continues with the discussion of how static branch clusters are determined. After providing more details on the methodology in Section 4, the results of clustered indexing are presented in Section 5 for various branch predictors. Section 6 discusses related work and in Section 7 we conclude the paper.

2. Clustered indexing

The aim of clustered indexing is to structure the information kept in branch prediction tables. At the same predictor budget, this information structuring process results in higher branch prediction accuracies. Otherwise stated, the cost-effectiveness of branch predictors can be improved when branch prediction is guided by additional cluster information. This section deals with various ways for using cluster information in a predictor, and the next section details on how branches can be divided in branch clusters.

Cluster information provides an alternative information source that is useful for indexing a structure. As branches within a cluster have some behavioral properties in common, we can decide to use a specific subset in the prediction tables per cluster of branches. This means that a given prediction table is partitioned into a set of smaller tables or *subtables*, each of those only used by a predefined cluster of branches. Fig. 1 illustrates different methods to use the cluster information in the indexing process. The term 'original index' is left unspecified as the technique is generally applicable in every prediction scheme. We compare the original indexing scheme (left) against clustered indexing where the cluster information is used as subtable identifier (center). Since identifying an entry within a subtable requires less bits compared to the index for the entire prediction table, the cluster information is used in exchange for some bits in the original index. Although smaller prediction tables increase aliasing, the overall performance would not degrade if the branches in a same subtable are chosen so that they rarely negatively interact. One of the contributions of this paper is to cluster branches such that using the cluster information in a setup with subtables augments the prediction accuracy by eliminating destructive aliasing.

Another approach to incorporate the cluster information is shown in the right of Fig. 1. In this case, the original index and the cluster identifier are hashed together to select an entry in the prediction table. This hashing strategy preserves the original index length, which is important to explore correlation in e.g., long histories. In the next section we discuss the identification of such branch clusters.

3. Identifying static branch clusters

In this section, we describe our experimental setup to obtain clusters of static branches that can share prediction state without degrading prediction accuracy. The identification of branch clusters is based on the following two observations. First, branches with identical time varying outcome behavior can use the same table section while benefiting from constructive aliasing (e.g., A and B in Fig. 2). Second, two branches sharing resources yet executing in strictly separable time intervals do not influence each other, except for some initialization period (e.g., C and D in Fig. 2). To incorporate both the outcome behavior and the time notion, we have chosen to quantify the behavioral properties of static branches by *the taken rate behavior in time slices*.

Our results indicate this is an appropriate metric, although we do not claim this is the best possible one. We have also analyzed the transition rate but it turned out to perform worse then the taken rate. The reason is that the transition rate distribution sharply peaks at 'no

A	1 1 1 1	0 0 0 0	1 1 1 1	0	1 0 1 1 1
B	1 1 1 1	0 0 0 0	1 1 1	1 0 0 0 0	1 1 0 1
C			0 0 0	0 1 1 1	1
D	1 1	1 1			

time →

A	100%	0%	100%	0%	80%
B	100%	0%	100%	20%	75%
C	NE	NE	0%	75%	100%
D	100%	100%	NE	NE	NE

Fig. 2. Up, branch outcome behavior for 4 static branches A, B, C and D with '1' and '0' for a taken and not-taken branch outcome, respectively. Down, representation by means of time varying taken rate behavior; NE, 'not executed'.

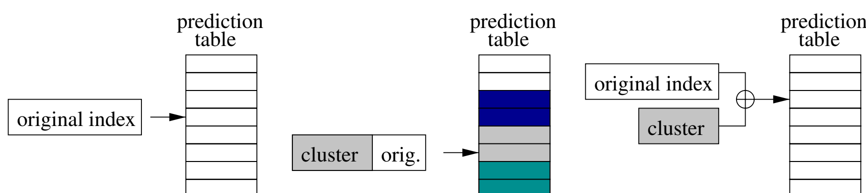

Fig. 1. Original indexing (left) compared to clustered indexing with subtables (center) and clustered indexing with hashing (right).

UNCORRECTED PROOF

Please cite this article as: Veerle Desmet et al., Clustered indexing for branch predictors, Microprocessors and Microsystems (2006), doi:10.1016/j.micpro.2006.08.003

transitions' and thus strongly taken branches are frequently clustered together with strongly not-taken branches. The combination of transition rate and taken rate might be an interesting choice, yet the purpose of this study is to explore a broad range of branch predictors to motivate which prediction schemes (bimodal, global, gshare, perceptron, etc.) benefit from a systematic indexing strategy. Fig. 2 illustrates how the time varying taken rate behavior is determined from the branch execution profiles. In the example, branches A and B can use the same entry without harming each other since A and B behave very similar. On the other hand, branches C and D interleave in time and can be mapped to the same entry. Whereas the example uses a time slice of 10 branches, our experiments were conducted with time slices of 10 M conditional branch executions. This time slice choice roughly corresponds to the interval length of 100 M instructions used by SimPoint [14] for detecting program phases. Based on execution traces, we get a vector of taken rate numbers for each static branch. Each of these numbers represents either the taken rate or a special `NE`-value (not executed or "do not care") denoting that the branch is not executed during that specific time slice.

Given the behavioral properties of branches – represented by their taken rate vector – we now consider the formation of clusters of branches with similar behavior. Therefore, we use the k-means clustering algorithm [12]. Starting with a random selection of k different cluster centers, the algorithm iteratively calculates the distances between all static branches and the cluster centers, and it associates each static branch with the cluster at minimal distance. Among all branches in a cluster the method redefines new cluster centers, which serve as initial cluster centers in the next iteration of the algorithm. We iterate this process a 100 times and we only consider stable solutions.

In the k-means algorithm, the distance from a branch to another branch (or cluster center) is defined to be the sum of the squared differences between the taken rates. It also takes into account the special `NE`-value that has zero distance to any valid value of the taken rate. Further, we associate a weight with each static branch according to its dynamic execution frequency. The complexity of the k-means algorithm is $\mathcal{O}(kN)$ per iteration where N is the number of static branches.

The k-means clustering is repeated for values of k (number of clusters) up to 50, each time evaluated for 20 different cluster initializations. To compare and pick out the best solution for the number of clusters we use the Bayesian information criterion, a metric for quantifying the goodness of a clustering given the data that has been clustered [12]. Hence, for SPECint2000, the number of clusters varies between 8 and 33, see Table 1. For all benchmarks, most branches populate clusters with centers that represent a rather constant taken rate behavior. Two important examples are the ever recurring clusters 'always taken' and 'always not taken', both reflecting that many branches have a tendency to be most of their time either taken or not taken.

Table 1
Benchmarks, inputs, the number of static branches executed, the number of clusters obtained by static branch clustering for reference and train input

Program	Ref input	Stat. branches	#Clust. Ref input	#Clust. Train input
164.gzip	random	150	26	24
175.vpr	route	598	12	38
176.gcc	scilab	14,880	33	30
181.mcf	ref	346	8	37
186.crafty	ref	1396	19	13
197.parser	ref	1974	33	22
252.eon	kajiya	281	21	20
253.perlbmk	splitmail 535	2055	30	42
254.gap	ref	566	20	27
255.vortex	lendian2	1603	15	18
256.bzip2	program	339	12	4
300.twolf	ref	569	23	26

4. Methodology

The experimental results in this paper are obtained for the SPECint2000 benchmarks. Except for *eon*, these benchmarks are compiled by the Compaq C compiler version V6.3-025 with optimization flags `-arch ev6 -fast -O4`; *eon* is similarly compiled by the Compaq C++ compiler version V6.3-002. For each of the benchmarks, we have skipped the first 50 M conditional branches (i.e., roughly half a billion instructions) to exclude program initialization code [15] and measure during the following 250 M conditional branches. Simulations are done in SimpleScalar [1] and unless otherwise mentioned, all results show the arithmetic average over the 12 benchmarks in Table 1.

5. Evaluation

Here, we use the branch clusters identified using profile information and we distinguish *self-clustered* from *cross-clustered*. Self clustering identifies branch clusters based on the reference input, whereas cross clustering uses the clusters based on the training input (unseen branches are randomly assigned a cluster number from the range between the maximum cluster number and the next power of 2). As all evaluations are done on reference traces, the self-clustered is somewhat optimistic while cross-clustered gives more realistic results. In general, it is thus expected that cross-clustered performs slightly worse then self-clustered. Every branch belongs to exactly one cluster, labeled by its cluster identification number. We assume that this information is made available to the architecture e.g., through the ISA by using hint bits [4] or a scheme similar to Jiménez [6].

Selecting a subtable is the most obvious use of the cluster identification number. It divides the original prediction table pro rata the number of clusters determined during

UNCORRECTED PROOF

branch clustering (see Table 1). Each subtable is equal in size and chosen to be a power of 2 in the number of subtable entries. Consequently, some table entries are left unused depending on the number of clusters.

5.1. Gshare

To start with we consider a gshare branch predictor [10] that computes the XOR of the branch address and the global history register to index the branch prediction table. We compare gshare's original indexing to clustered indexing in a subtable setup. Fig. 3(a) shows that clustered indexing outperforms the originally indexed gshare for bit budgets up to 64 Kib. For a 8 Kib gshare the misprediction rate is decreased from 7.3% downto 5.9%, a reduction of the misprediction rate by 19% in case of self-profiled clusters. Cross-clustered attains 5.4% and preserves 12.3% reduction of the misprediction rate at 8 Kib, and a 4 Kib cross-clustered gshare reaches 7.3%. In other words, with cross-clustered indexing we can halve the predictor budget while performing equally well. For large predictor sizes (detail in Fig. 3(b)) clustered indexing with subtables cannot improve much. This is because larger prediction tables do not suffer from aliasing anyway and the correlation from long histories is lost due to subtabling (see Section 5.1.1).

Fig. 3(c) compares a global predictor scheme (GAg [17] which merely uses the global history register to index a prediction table) against a gshare to prove that both schemes are equivalent when clustered indexing is applied. This means that the additional benefit of XORing the branch address with the global history, as performed in a gshare predictor, is nullified when applying clustered indexing. For large predictor sizes (see Fig. 3(d)) gshare keeps a small benefit over a global prediction scheme.

In addition, Fig. 3(e) shows that for small sizes a clustered gshare outperforms three more powerful predictors namely bi-mode, gskew and a hybrid predictor. The bi-mode prediction scheme [9] has three predictors: one bimodal and two gshares. The counter read from the bimodal table chooses which of the two gshare's will be used for prediction. In a sense, the bi-mode selects between two clusters based on the bimodal predictor. The gskew prediction scheme [11] uses three gshare predictors – each of them indexed by a different indexing function – and the majority vote is used as prediction. The idea behind gskew is that it is unlikely that destructive aliasing occurs in more than one predictor at the time. The hybrid scheme is similar to the EV6 branch predictor [7] which uses different sources of information to capture different branch behaviors. The global/local hybrid predictor uses a bimodal predictor to choose the final prediction between a local or global prediction component.

Fig. 3(e) shows that up to 8 Kib bi-mode, gskew and hybrid have misprediction rates that are comparable to an original gshare. This is because small budgets cannot justify the cost for the meta-predictor that dynamically selects between subpredictors. Yet, small predictors can benefit from profile-based systematic clustering while avoiding the cost of a meta-predictor. It is interesting to point at a clustered version of the bi-mode predictor – only the bimodal predictor clustered – performs worse than a clustered gshare. The weaker effect of clustering in the bi-mode predictor might be related to the interaction with its partial update policy.

5.1.1. Why clustered indexing works

Having dealt with the positive impact of clustered indexing on misprediction rate, we explain why it is well-performing. Hereto, we compare the alias rate between an original and self-clustered gshare.

We associate to each prediction its *identity*. This identity is an unhashed version of the bits used to index the prediction table. For example, the gshare identity concatenates the used bits from branch address as well as from global history. To detect aliasing, we store the identity in the prediction table next to the information that has been updated for that specific identity. Aliasing occurs when the stored identity does not match the identity of the prediction being made. Further, we break down this alias fraction into (i) *destructive* aliasing representing the predictions leading to a misprediction, (ii) *constructive* aliasing for correct predictions, and (iii) *neutral* aliasing when the prediction equals the prediction of an alias free predictor that uses the original identity as index.

In Figs. 3(g) and (h), we see that slightly more aliasing is introduced in a predictor that uses clustered indexing. However, as stated in the introduction, the results illustrate that clustered indexing benefits from turning destructive into neutral aliasing. Especially for small predictor sizes where aliasing occurs frequently, this mechanism augments prediction accuracy, whereas for large predictor sizes – where little aliasing is measured – clustered indexing could not benefit from eliminating destructive cases. In the next subsection, we discuss the reasons why clustered indexing is not working in large prediction tables and we present an alternative method to incorporate cluster information.

5.1.2. Hashing as alternative to subtables

Side effect of using subtables is the smaller portion of the original index being used (smaller tables require less index bits), which also explains the slightly higher alias rate. In case of gshare, subtabling shortens the effective length of the global history. Moreover, the setup with subtables assigns $\frac{1}{k}$ number of entries to each of the k clusters. In large predictors, this uniformly sized subtables are wasting entries for clusters that are not densely populated. Therefore, we present an alternative mechanism in which cluster information is incorporated in the hashing function in a similar way as other information sources.

As alternative to subtables, we propose *hashing*, i.e., the concatenation of the cluster identification number with the branch address after which this whole bit string is XORed with the global history. This is in contrast

UNCORRECTED PROOF

Please cite this article as: Veerle Desmet et al., Clustered indexing for branch predictors, Microprocessors and Microsystems (2006), doi:10.1016/j.micpro.2006.08.003

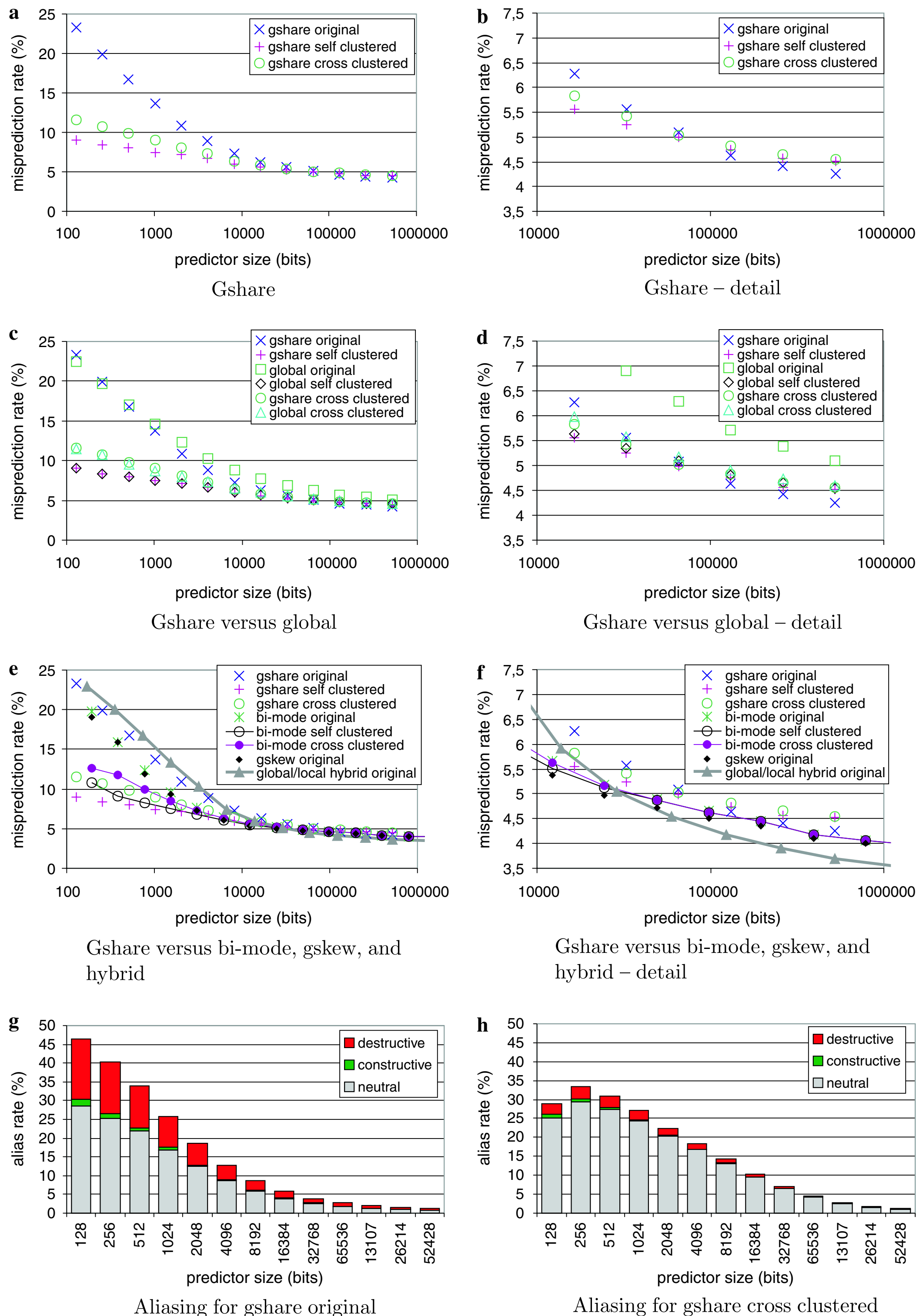

Fig. 3. Clustered indexing for gshare.

with the previous section where the branch address and the global history are XORed after which the cluster identification is concatenated. Hashing does not need subtables, so the entire table is indexed which allows the use of histories as long as in the original predictor.

Fig. 4 illustrates that for large predictor budgets, hashing with the cluster identification number can reduce the misprediction rate by 5% on average for a 2 Mib gshare predictor (from 4% downto 3.8%). For those large budgets it follows that only half the hardware budget is required to obtain the same performance as the original gshare scheme.

Please cite this article as: Veerle Desmet et al., Clustered indexing for branch predictors, Microprocessors and Microsystems (2006), doi:10.1016/j.micpro.2006.08.003

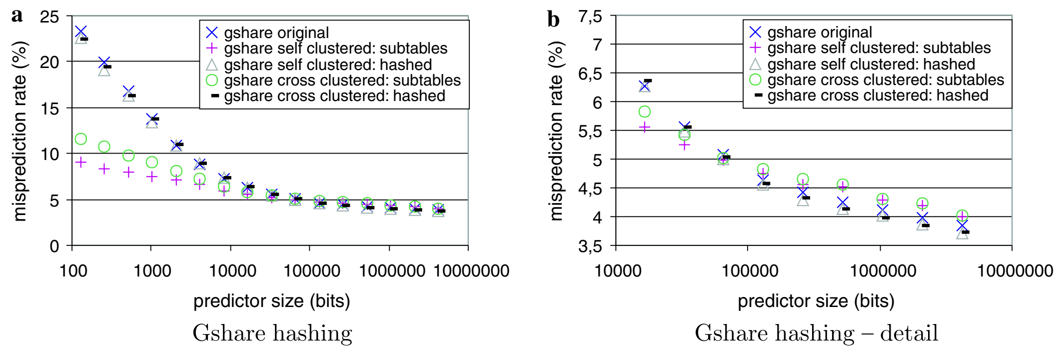

Fig. 4. Gshare, hashing of cluster identification number versus using subtables.

5.2. Other prediction schemes

Fig. 5 demonstrates the use of clustered indexing with subtables for various other branch predictor schemes.

For the bimodal prediction scheme, a prediction table is indexed by the lower order bits of the branch address. The selected entry contains a 2-bit saturating counter from which the upper bit is used as prediction. We point at the fact that even extremely small bimodal predictors with clustered indexing (e.g., the left most point in Fig. 5(a) is 128 bit) achieve similar accuracies as the biggest original bimodal predictors.

A perceptron branch predictor [5] is a simple one layer neural network. The network output serves to decide whether the branch is predicted taken (output ≥ 0) or not-taken (output < 0). This output is computed as the dot product of a *perceptron* vector and an input vector. The perceptron vector is taken from a table which is indexed by the lower order address bits. The input vector consists of global history bits in which taken and not-taken branches are represented by +1 and −1, respectively. We evaluated a wide range of perceptron predictors varying the global history length from 4 to 40 in steps of 4 as well as varying the number of perceptrons from 32 up to 512 by powers of 2. Fig. 5(c) shows the best configurations among all the evaluated perceptron designs. A slice of 5 points in Fig. 5(d) corresponds to a fixed number of perceptrons. We observe that even a very accurate 8 Kib perceptron predictor improves 29% (self) and 7% (cross) over the original scheme by using clustered indexing (from 7.1% downto 5% and 6.6%, respectively). Otherwise stated, an 8 Kib perceptron predictor can be replaced by a clustered perceptron version that achieves 6.9% while requiring only 4.3 Kib, which means a saving of nearly half the predictor budget.

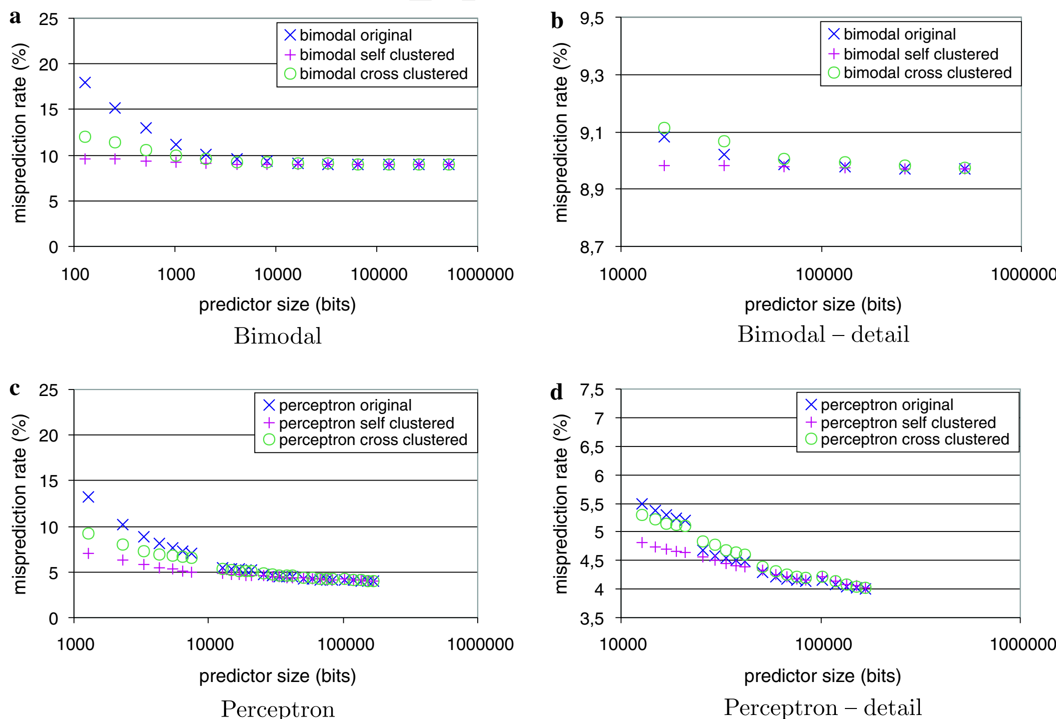

Fig. 5. Clustered indexing with subtables for various branch prediction schemes.

At 20 Kib, the misprediction rate is still reduced by 11% (self) and 2% (cross) when using clustered indexing. Starting from 40 Kib we measure 4% mispredictions for both original and clustered indexing. So, clustered indexing improves a perceptron predictor until the point where the misprediction rate saturates.

The local branch predictor has 2 tables. The first table is indexed by the lower part of the branch address and captures the per-address (or local) history. The latter history is then used to index the second table. Since the local predictor consists of 2 tables, clustered indexing can be applied to either or both of the tables. Besides the original local branch predictor for which we explore various configurations, we also evaluated (results not shown) the clustering variants at either of the tables as well as performing clustered indexing at both tables. The conclusion is that independent on the cluster level, small local history predictors clearly benefit from clustered indexing. However, the work by Yeh and Patt [17] defines clusters (they say sets) of branches based on branch address bits, whereas we concentrate on behavioral characteristics. We found that the use of address bits as cluster identifier performs only slightly better (from 10.9% to 10.3% at 2 Kib) than the original predictor, whereas clustering reduces the misprediction rate from 10.9% downto 6.6% at the same hardware budget.

In conclusion, subtables are well-performing for a variety of small predictors. For large predictors, hashing with cluster information can exploit the huge amount of resources in a more clever way than subtables.

Finally, Fig. 6 compares the results by individual benchmark for gshare and perceptron branch predictors at 8 Kib. Although the average is promising, for some benchmarks clustered indexing increases the misprediction rate. In the self-clustered setup this occurs only for *parser*, and this is because we apply a conservative approach in the budget for each cluster: e.g., to fit the 33 clusters of *parser*, we allocate 64 (power of 2 greater than 33) subtables and use only 33 of them. This means nearly half the predictor table is left unused in the clustered setup, i.e., the results are fair and are even better than presented if we properly calculate the used predictor size. In the cross-setup – where the phenomenon occurs more often – there is an additional factor due to the unseen branches during the profiling step, e.g., *perlbmk* behaves completely different under test/train input compared to ref input.

5.3. Alternative cluster assignments

Until now, the number of clusters can be relatively large and varies between the benchmarks (see Table 1). In this section we evaluate alternative cluster assignments where the number of clusters is fixed to 2, 3, 4, 8 or 16 clusters, respectively. We will first show the results of cross-profiling that illustrate how much misprediction rate can be expected in a realistic setup with a fixed number of clusters. After this, we demonstrate that if clustered indexing is used the worst case scenario is similar to the original indexing schemes.

Fig. 7 illustrates the misprediction rate for the individual benchmarks for a predefined number of clusters for gshare. It follows from the figure that a small number of clusters (2, 3 or 4) outperforms the original gshare scheme for all the benchmarks. On average, an optimum in the cluster number is reached for 4 clusters. This requires only 2 bits of branch hints to encode which is far more interesting from a hardware perspective. Note that an interesting trade-off exists for 3 clusters, as accuracy is comparable to the case of 2 and 4 clusters, while predictor size is reduced by 25%.

Fig. 8 shows the results for the worst case scenario in which branches are assigned to clusters in a random way. We use random assignment of branches to clusters to model the worst case scenario where profiling information is very inaccurate. In practice, branch directions can be profiled with high accuracy [3] and profile information is transferable among several inputs to the same program [13]. However, we want to show that clustered indexing does not degrade performance, even when using the worst possible profiling information. The results show that in the worst case clustered indexing equals the original prediction scheme for each of the predictors. Yet, it is realistic that randomly assigned clusters outperform the original scheme, e.g., gshare with 16 random clusters is better than the original indexing strategy in the small budget range. Similar conclusions can be drawn for the hashing strategy for the large budget ranges. In conclusion, clustered

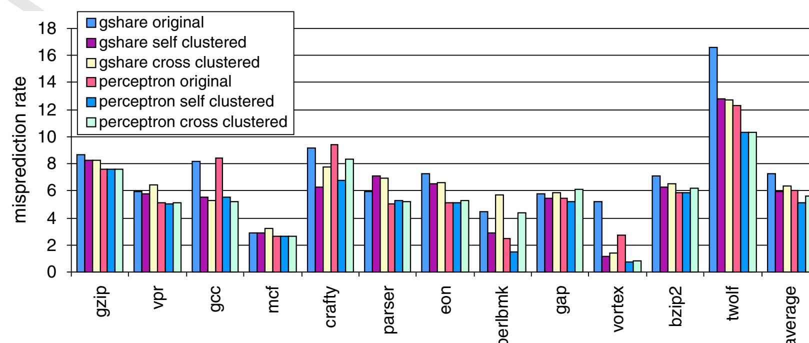

Fig. 6. Comparison by individual benchmark at 8 Kib.

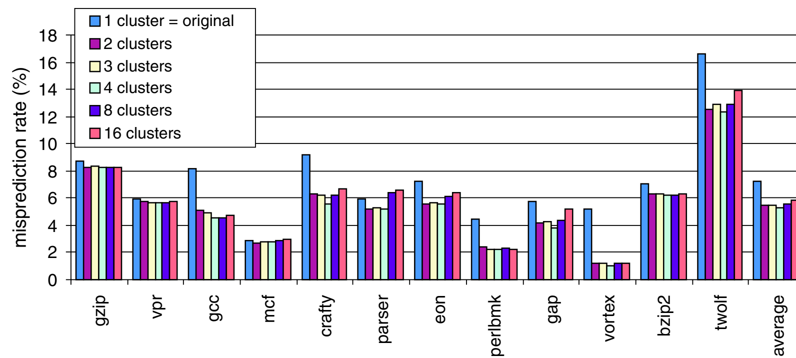

Fig. 7. Comparison by individual benchmark at 8 Kib for a fixed number of clusters.

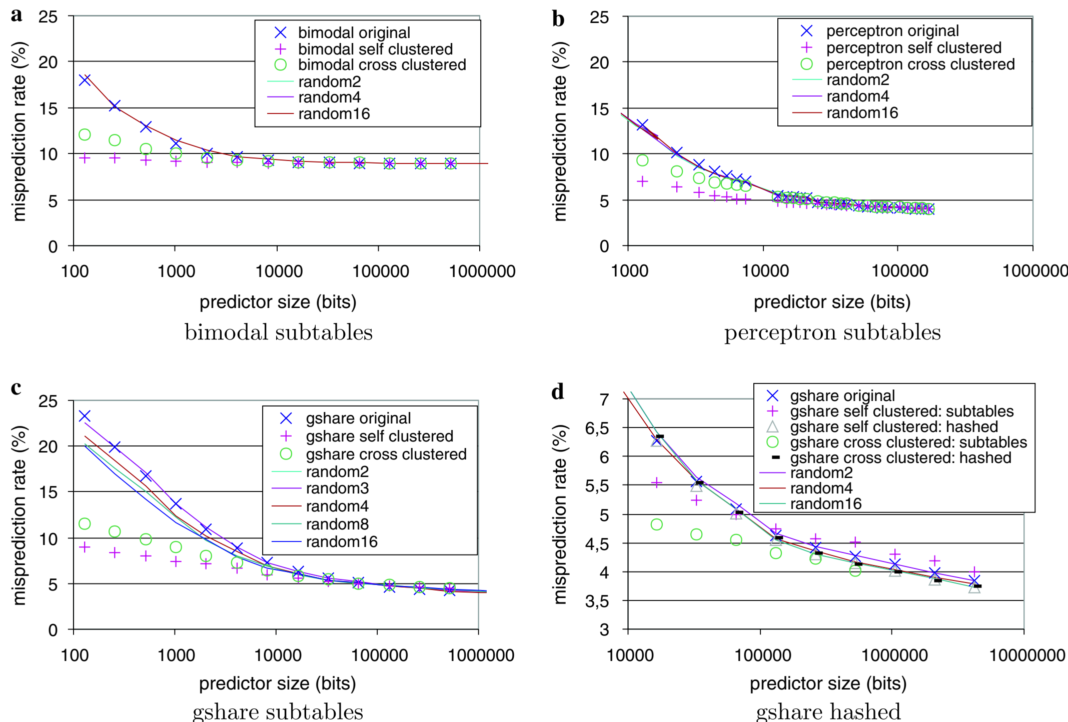

Fig. 8. Worst case scenario: branches are randomly assigned to clusters; in this case clustered indexing equals the original indexing scheme.

indexing can be applied safely: when profile information is wrong, the prediction accuracy will not be degraded.

5.4. IPC

Finally, we demonstrate the performance of clustered indexing in a realistic processor setup. The processor parameters used in these experiments are summarized in Table 2.

For branch predictors, we compare an original gshare with a clustered indexed gshare with 4 clusters (subtables) determined with cross-profiling. Fig. 9 shows the average number of instructions that are committed per cycle (IPC) for 4 different benchmarks, namely `gcc`, `crafty`, `parser` and `bzip2`. Up to 64 Kib predictor sizes we measured speedups. Often these speedups are more than 10%. At 8 Kib we have 27%, 18%, 4% and 5% speedup for `gcc`, `crafty`, `parser` and `bzip2`, respectively.

6. Related work

Kim and Tyson [8] presented a profile-based analysis of working sets for branch prediction. They partition branches in different working sets when they interleave during execution, i.e., purely based on the timing aspect. In addition to their temporal exploitation, we also rely on similarity in outcome behavior while benefiting from neutral interaction between branches. Although Kim and Tyson focus on the characteristics of the working sets, these sets can be used for indexing tables too because branches within different sets can not suffer from aliasing. Their evaluation only includes a local branch predictor, whereas we present the

Table 2
Baseline processor configuration simulated by *sim-outorder*

Parameter description	Setting
Fetch	Up to 4 instructions per cycle, 2 taken branches
BTB	4096 entries, 2-way
RAS	64 entries, top-of-stack pointer recovery
Misprediction penalty	14 cycles, 2 cycles on misfetch
Decode/issue/commit	Up to 8 instructions per cycle
Instruction window	128
Load/store queue	64
Functional units	8 INT ALU's, 2 INT mult/div, 2 FP ALU's, 2 FP mult/div
Memory ports	2
L1 data cache	64 KB, 2-way, 32 B blocks, LRU 1 cycle hit latency
L1 instruction cache	64 KB, 2-way, 32 B blocks, LRU 1 cycle hit latency
L2 unified cache	8 MB, 4-way, 64 B blocks, LRU 6 cycles latency
Memory	151 1
TLB	128 entries, fully associative

488 use of more intelligent indexing techniques to a range of
489 branch predictors.
490 Chang et al. [2] suggested branch classification to
491 optimize hybrid branch prediction components while
492 associating each branch with the component best suited
493 to predict its direction. The class of branches is deter-
494 mined by their overall dynamic taken rate collected dur-
495 ing program profiling. Their branch classification can be
496 considered as the special case with one time phase.
497 Instead of tuning components of a hybrid branch predic-
498 tor, we describe a general technique for efficiently using
499 existing prediction hardware.
500 Yeh and Patt compared two level schemes in general
501 [17], and they introduced the notion of per-set predictors.
502 In our terminology these sets are subtables, yet they use
503 branch address bits to select the set. More precise, they
504 put branches in a block of 256 instructions in the same
505 set and rely on the fact that beyond these boundaries
506 branches occasionally interleave.
507 The bi-mode predictor [9] dynamically divides branches
508 into two groups according to the outcome of a choice pre-
509 dictor. They further use global history to identify the final
510 prediction in the selected group (implemented as a separate
511 table), and a partial update strategy is managed. Our
512 experiments regard the partitioning of branches in more
513 than 2 groups.
514 Skadron et al. [16] categorize different sources of mis-
515 predictions. They show that aliasing is an important
516 source of mispredictions in various predictors schemes.
517 They also pointed to the importance of small branch
518 predictors.
519 Jiménez [6] describes a possible way for propagating
520 cluster information to the architecture without modifying
521 the ISA. Hereto, the code layout is changed such that the
522 program counter encodes the cluster information. The
523 clusters are chosen based on profiling whereby branches
524 are classified into 4 predefined behavior classes. The evalu-
525 ation is done for a Pentium 4, i.e., with a fixed size branch
526 predictor only.

UNCORRECTED PROOF

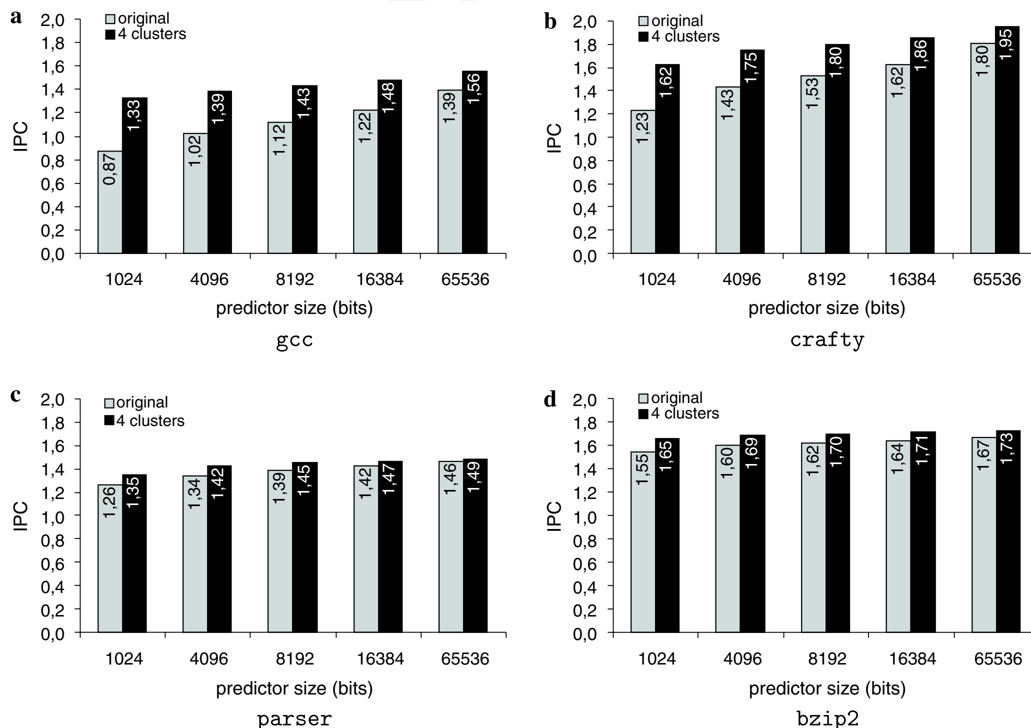

Fig. 9. IPC comparison for an original and a clustered indexed gshare predictor.

Please cite this article as: Veerle Desmet et al., Clustered indexing for branch predictors, Microprocessors and Microsystems (2006), doi:10.1016/j.micpro.2006.08.003

7. Conclusion

This paper introduces clustered indexing, an new indexing mechanism that eliminates destructive aliasing by systematically selecting clusters of branches that share prediction information. These clusters of branches are identified by a *k*-means cluster algorithm based on the time varying taken rate behavior of a branch profile.

We have shown that in various dynamic branch predictors, clustered indexing with subtables can significantly reduce misprediction rates in small predictors; larger predictor sizes benefit from hashing the cluster identification in exchange for less branch address bits. For SPEC CPU2000 INT, this profile-guided mapping between branches and table entries reduces 12.3% of the mispredictions in an 8 Kib gshare predictor. At this budget a clustered indexed gshare outperforms bi-mode, gskew, perceptron as well as a global/local hybrid scheme. Even a very accurate 8 Kib perceptron predictor improves 7% over the original scheme by using clustered indexing. Otherwise formulated, the same misprediction rates is obtained with 2–4 tumes less hardware on small budgets, and half the budget on larger predictor sizes. We also demonstrated that a worst case scenario for clustered indexing equals the original prediction performance. According to the results in this paper, the use of well-formed clusters of branches suggests large improvements in prediction accuracies (as well as in speedup) for all kinds of dynamic branch predictors.

Acknowledgements

Veerle Desmet is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). Hans Vandierendonck is a postdoctoral researcher of the Fund for Scientific Research-Flanders (FWO). This research was also funded by Ghent University.

References

[1] D. Burger, T. M. Austin, S. Bennett, Evaluating future microprocessors: the SimpleScalar Tool Set, Technical report, Computer Sciences Department, University of Wisconsin-Madison, July 1996.

[2] P.-Y. Chang, E. Hao, T.-Y. Yeh, Y. Patt, Branch classification: a new mechanism for improving branch predictor performance, in: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 22–31, November 1994.

[3] J.A. Fisher, S.M. Freudenberger, Predicting conditional branch directions from previous runs of a program, in: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 85–95, October 1992.

[4] Intel Corporation. IA-32 Intel Architecture Software Developers Manual Volume 1: Basic Architecture, 2001. Order Number 245470.

[5] D.A. Jiménez, Neural methods for dynamic branch prediction, ACM Transactions on Computer Systems 20 (4) (2002) 369–397.

[6] D.A. Jiménez. Code placement for improving dynamic branch prediction accuracy, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, pp. 107–116, 2005.

[7] R. Kessler, E. McLellan, D. Webb, The Alpha 21264 microprocessor architecture, in: Proceedings of International Conference on Computer Design, pp. 90–105, October 1998.

[8] S.P. Kim, G.S. Tyson, Analyzing the working set characteristic of branch execution, in: Proceedings of the 31st Annual International Symposium on Microarchitecture, pp. 49–58, November 1998.

[9] C.-C. Lee, I.-C.K. Chen, T.N. Mudge, The bi-mode branch predictor, in: Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 4–13, December 1997.

[10] S. McFarling, Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[11] P. Michaud, A. Seznec, R. Uhlig, Trading conflict and capacity aliasing in conditional branch predictors, in: Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 292–303, June 1997.

[12] D. Pelleg, A. Moore, X-means: extending k-means with efficient estimation of the number of clusters, in: Proceedings of the Seventeenth International Conference on Machine Learning, pp. 727–734, 2000.

[13] S. Savari, C. Young, Comparing and combining profiles, Journal of Instruction Level Parallelism 2 (2000).

[14] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, IEEE Micro 23 (6) (2003) 84–93.

[15] K. Skadron, Characterizing and removing branch mispredictions, June 1999.

[16] K. Skadron, M. Martonosi, D.W. Clark, A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions, in: Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques, October 2000.

[17] T.-Y. Yeh, Y.N. Patt, A comparison of dynamic branch predictors that use two levels of branch history, in: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 257–266, May 1993.

[18] C. Young, N. Gloy, M.D. Smith, A comparative analysis of schemes for correlated branch prediction, in: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 276–286, May 1995.

UNCORRECTED PROOF

Please cite this article as: Veerle Desmet et al., Clustered indexing for branch predictors, Microprocessors and Microsystems (2006), doi:10.1016/j.micpro.2006.08.003