# Performance Enhancements for Superscalar processors

Gogul Balakrishnan, Joseph Stanley
{bgogul,jass}@cs.wisc.edu

11th May 2002

## Abstract

Superscalar processor design has been the standard for building good performance computers. This project aims at investigating enhancements to a base superscalar architecture to further improve the performance. The project consists of two main phases. A limited resource availability is assumed (32k of RAM to be distirbuted among L1 caches, branch predictors, BTB, enhancements). With this assumption we try to achieve the best possible performace by tweaking cache and branch predictor configuration. In the second phase, the baseline architectiure is extended with performance enhancement features. The total RAM available for the enhancements is again within the 32K.

## 1 Introduction

In order execution imposes unreasonable control dependencies on the execution stream. The data dependencies are the those which constrain the order of execution. Superscalar processors, that try to execute the instructions in the order of data dependencies, has been the standard for building high performance computers. Among others, the branch miss-prediction rate and cache miss-rates are the main factors that limit the performance of a superscalar processor. This project aims at investigating these factors and enhancements to offset these factors. A limit of 32K was imposed on the amount of RAM that can be used for level 1 cahes, BTBs and the enhancements. The project was divided into two main phases.

The first phase was aimed at achieving the best possible configuration within the imposed limit of 32K. The various parameters that can be tweaked are the L1 cache sizes, block sizes of the caches, associativity and the type and configurations of the branch predictor. The IPC obtained in the first phase is used as a basis for comparing the enhacements implemented in the second phase.

In the second phase, again within the imposed limit of 32K, enhacements to improve branch prediction accuracy and cache hit-rates are investigated. Since the RAM for the enhancements are taken from the baseline configuration, the comparsion of performance between the enhanced and baseline configuration shows the improvement that can be achieved through enhancement features rather than extended cache.

## 2 Simulation environment

In-order to study the performance of a superscalar processor, the SimpleScalar simulation tools were used. SimpleScalar simulates an out of order superscalar processor. It is possible to simulate three memory hierarchies including two levels of cache. SimpleScalar also implements a simple bimod branch predictor, two level and combined branch predictor. There are certain caveats in using Simplescalar though. It should be noted that it uses infinite store and write buffers and caches very similar to lockup-free caches. The performance seen in the simulations may not be representative of actual implementations.

Table 1: **Benchmarks**

| Benchmarks | Input | #Inst |
|---|---|---|
| perl.ss scrabbl.pl | scrabbl.in | 46M |
| cc1.ss | gcc.i | 53M |
| compress95.ss* | compress.in | 100M |
| go.ss 9 9 | - | 164M |

*compress95 is fast forwarded by 400M inst.

| Cache | Size | BSize | # Sets | Assoc | Tags |
|---|---|---|---|---|---|
| L2 | 128K | 128 | 256 | 4 | |
| DL1 | 8K | 32 | 32 | 8 | 704 |
| IL1 | 16K | 128 | 4 | 32 | 368 |
| Total | **24K** | | | | **1072** |

Note: All caches use LRU replacement policy

Table 2: **Cache Configuration**

### 2.1 Benchmarks

A subset of SPECint95 benchmarks; go, vortex, perl and compress were used. Unweighted harmonic mean of IPCs was used to compare the overall performance of the enhanced processor against the baseline configuration. The number of instructions simulated in each benchmark and their corresponding inputs is tabulated in Table1.

## 3 Phase 1

### 3.1 Resource distribution

A 4-issue superscalar processor was simulated. The baseline configuration consisted of 32K RAM to be distributed among the level 1 caches, branch predictors and BTB. The level 2 cache was fixed at 128 KB with a maximum associativity of 4. The size of TLBs were taken to be big enough that they did not affect the performance of the processor. Only one read and one write port to the memory was assumed.

The distribution of RAM is tabulated in tables 2, 3, 4 and 5. Except where indicated explicitly the sizes are indicated in bytes. The return address stack size was set to 8.

**Combined Predictor**

| Component | Configuration | Size |
|---|---|---|
| Meta table | 1024 | 256 |
| Bimode | 2048 | 512 |
| Two level | L1-1 (global history) | 13 bits |
| | L2 - 8192 | 2048 |
| | | **2818** |

Table 3: **Branch Predictor Configuration**

| Number of sets | 128 |
|---|---|
| Associativity | 4 |
| Size | 2 K |
| Tags | 1632 bytes |

Table 4: **BTB Configuration**

### 3.2 Tradeoff decisions

A number of factors affect the IPC of a processor. The cache miss rates, branch predictor accuracies, the window sizes, number of functional units are a few among them. The degree to which each one affects the IPC also varies. In the first phase the only factors that could be experimented are the cache and branch predictor configurations. Hence the natural aim was to reduce the cache miss-rates and improve the branch prediction accuracy.

The only limitation being the total RAM used, a wide range of configurations are possible for the cache and branch predictors. Investigating all the possible configuarations is very time consuming and is not necessary. Certain characteristics of the instruction and data stream can be used to prune the search space. A reasonable configuration was choosen and keeping that as a basis, the configurations were tweaked to better the performance.

| Total mem (w/o tags) | 29442 bytes |
|---|---|
| Tags | 2704 bytes |
| Total mem (w/ tags) | **32146 bytes** |

Table 5: **Memory usage**

### 3.2.1 Cache configuration

The parameters that affect the miss rates of a cache are the size, associativity and block size.

**Cache size** The only unit of a super scalar processor that executes in-order is the instruction fetch unit. It is in the critical path. So the impact of an instruction cache miss on the performance of a superscalar processor is more pronounced than that of a data cache. Hence more RAM was devoted to the instruction cache; 16 KB for the level 1 I-cache and 8 KB for the level 1 D-cache.

**Block size** In general the instruction stream has more spatial locality than the data stream. This implies that increasing the block size for the instruction cache can increase the hit rate.

On the other hand, increasing the block size of the cache indefinitely will also have a negative impact. As the block size increases the size of the data that is fetched into the cache or replaced from the cache on each miss increases. A very large block size will result in unwanted data being fetched and also replace data that is currently being used, resulting in cache pollution. This can be seen from the figures 1 and 2. With an increase in block size over 64 bytes the miss-rates increase for D-cache.

However in the case of I-cache this effect is not more pronounced. In the instruction cache the spatial locality is in terms of basic blocks/procedures, which are generally a few hundreds of bytes in size. This suggests that the block size of the instruction cache should be of the order of few hundreds of bytes. On the other hand, spatial locality in the data cache is in the order of few tens of bytes on an average and the block size should be limited to this size.

Another effect of the cache block size is the latency to fetch the block. With an increase in the block size the latency to fetch the block from the next higher level in memory also increases proportionately. This was evident in our simulations. We had two different block sizes of 128 bytes and 256 bytes for the L2 cache. The miss-rates were 0.1688 and 0.1601 respectively. However the IPC for the 256 bytes configuration was lesser than that of 128 even though the

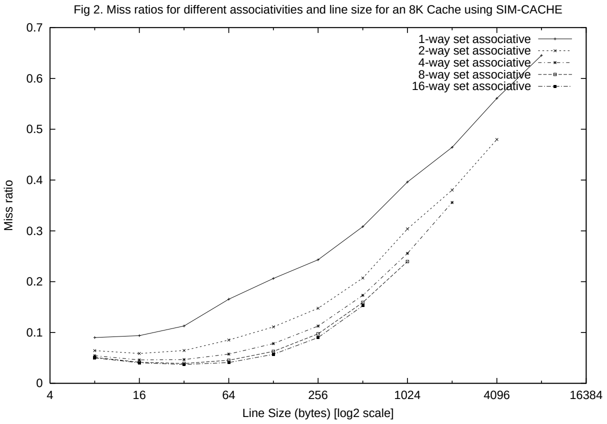


Figure 1: D-Cache miss-rates for different configurations

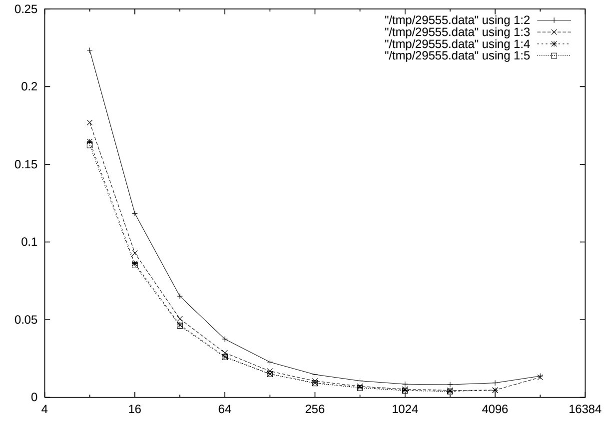


Figure 2: I-Cache miss-rates for different configurations

miss-rate was higher for the 128 case. This is due to the additional latency in fetching 128 more bytes from memory.

Thus a trade off between the block size /locality and latency/cache pollution exists; and an optimal block size taking these factors into account was found.

**Associativity** A set associative cache is aimed at reducing conflict misses. Increasing the associativity of a cache generally increases the hit rate. In a fully associative cache a block of data can go into any cache block which reduces the conflict misses to the minimal possible value for any given cache size. However increasing the associativities beyond a certain point does not help much. Also the cost and complexity of implementation increases with increase in associativity. From figure 1 we find that the optimal associativity for the D-Cache is 8.

### 3.2.2 Branch predictor configuration

As mentioned in 3.2.1, instruction fetch unit is in the critical path of the processor. To facilitate an increase in the fetch bandwidth a highly accurate branch predictor is necessary. There are different branch predictors with varying prediction accuracies using the same amount of hardware [EDEN 98]. Among the various branch predictors, 2-level predictors with gshare type lookup performs the best.

2-level predictors perform well because they capture the correlation between various branches through history of recently executed branches. 'gshare' type lookup reduces destructive interference between entries by distributing the entries uniformly across the branch predictor table.

**Branch Target Buffer** A BTB is a cache of predicted addresses which can be used to predict the address of a branch. Without a BTB a branch predictor is not very useful in a superscalar processor. So the effectiveness of the branch predictor is improved by having a good BTB.

Since BTB is a cache, all the tradeoffs discussed in 3.2.1 apply to the BTB . Since block size of BTB is fixed, one can vary the associativity and number of sets. We have chosen to have a 2K BTB with an associatvity of 8, which seems to perform well.

| Benchmark | IPC |
|---|---|
| cc1 | 0.8295 |
| compress95 | 0.7295 |
| go | 0.8242 |
| perl | 1.4226 |
| **Total** | 0.8904 |

Table 6: IPCs for different benchmarks

## 3.3 Results

Table 6 tabulates the IPCs obtained for the four benchmarks in the first phase. The overall IPC is also found to be **0.8904.**

# 4 Phase 2

From the experiments of the first phase, we were able to identify the spots which need to be improved.

One observation is that, except for perl, the branch miss prediction was 10%, with "go" having a much higher miss prediction of 20%. As explained in section 3, branch miss prediction penalty can have a very bad effect on the performance. Hence an enhancement we explored is an improved brance predictor. Also, fetch unit being in the critical path of the processor, a reduction in instruction cache miss-rate will improve the performance. Hence, instruction prefetching should also be incorporated.

One general observation was that i-cache miss rates was lesser than the d-cache miss rates. This might have been the artifact of a larger i-cache size. It may not be to our advantage to increase the size of d-cache at the expense of i-cache. So we investigated various ways of improving data cache performance. One obvious method is prefetching. We looked at various kinds of prefetching and chose the stirde prefetching scheme.

## 4.1 Indirect jump prediction

BTB is a cache that stores the target addressses of the recent branches. For an indirect jump the BTB stores the last target address of the jump instruction. However the target of the indirect jump can change with every instance of the jump. So the BTB will be mispredicting the target every time. In order to over come this, we can try to store all the targets of the jump and predict one of those based on the recent history.

'gcc' and 'perl' are interpreters and they are expected to have large number of indirect branches due to switch/case statements in source code. So improving the prediction of indirect jumps will be worthwhile.

In [CHAN 97], a prediction mechanism, the Target Cache, for predicting the targets of indirect jumps is introduced. The benchmarks that were considered, gcc and perl have a considerable number of indirect branches and will benefit from a better target prediction. Hence the Target Cache is used to enhance the accuracy of the baseline branch prediction mechanism.

### 4.1.1 Description

The target cache uses a concept central to the 2-level branch predictor; the branch history is used to distinguish among different dynamic occurences of each indirect branch. When fetching an indirect jump, the fetch address and the branch history are used to form an index into the target cache as shown in figure 3. The target cache is then accessed for the indirect jump. Later, when the indirect branch retires, the target cache is accessed again using the index and the computed target for the indirect jump is written into the target cache. As the program executes, the target cache records the target for each indirect jump encountered. When fetching an indirect jump, the target cache is accessed with the fetch address and the branch history to produce the predicted target address. The target cache improves on the prediction accuracy achieved by the BTB-based schemes for the indirect jumps by choosing its prediction from all the targets of the indirect jump that have already been encountered rather than just the target address that was most recently encoutered as is done by BTB.

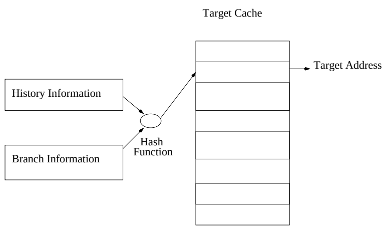


Figure 3: Target Cache Lookup

The Target Cache can be either tagged or tagless. Since the RAM was a limited resource, a tagless target cache is used. This will result in both constructive and destructive interferences which would have been avoided in a tagged cache. The performance difference between the tagged and tagless target caches for the benchmarks considered was not investigated. Even with tagless caches we can have a very good prediction accuracy by carefully choosing the indexing function.

### 4.1.2 Branch History

There are several ways of tracking the history of branches.

- Path history - Path history has been known to improve prediction accurary of branch prediction [NAIR 95]. In path history we will be tracking a subset of the bits of the jump targets. One can keep track of the history of all control instructions, branch or indirect jumps. Also the number of bits and position of bits used in the target address as a part of history information can be varied.
- Pattern history - In a pattern history we keep

track of the directions of the recent branches. Here too we can restrict the type of instructions that are being used as a part of history. For example, keeping pattern history of unconditional jumps will not help in distiguishing the dynamic instance of the current branch.

### 4.1.3 Performance enhancements

Some memory was allocated from the BTB to the target cache. A target cache with 512 entries was experimented with various history schemes.The results are shown in figures 4 and 5.

The results clearly show that keeping track of target of indirect jumps alone will help a lot. This is on the expected lines. 'gcc' and 'perl' are compilers/interpreters and they typically have an automaton which is coded as switch/case statements. Each state will be a target of the indirect jump. The next state depends on the past states in the automaton. So will the branch target address.

When we increase the number of bits that are tracked, we are reducing the number of branch instructions we are keeping track in history. So increasing the number of bits will reduce the prediction accuracy as is seen in the graphs.

Changing the bit position actually reduces the accuracy. This is also expected. Normally jumps are short. So if we keep track of the higher order bits, the PC of the last n branch instructions would have had the same pattern and so we would not be able to distinguish between them.

The improvements we gained for IPCs by adding the target cache are summarized in figure 6. The third bar for each benchmark shows the percentage improvement over the baseline configuration.

## 4.2 Instruction prefetching

Instruction prefetching can effectively reduce the instruction cache misses, thus improving performance. Instruction prefetching speculatively brings the instructions needed in the future. If the instruction prefetching scheme can predict future instructions accurately and bring them in advance, most of the I-Cache misses can be eliminated. Since the branch

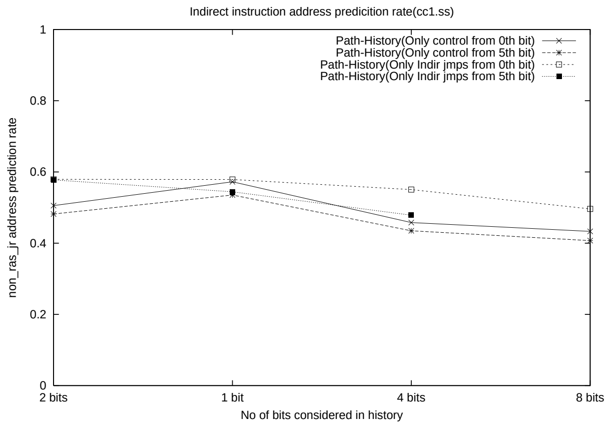


Figure 4: Target cache prediction accuracy for cc1.ss

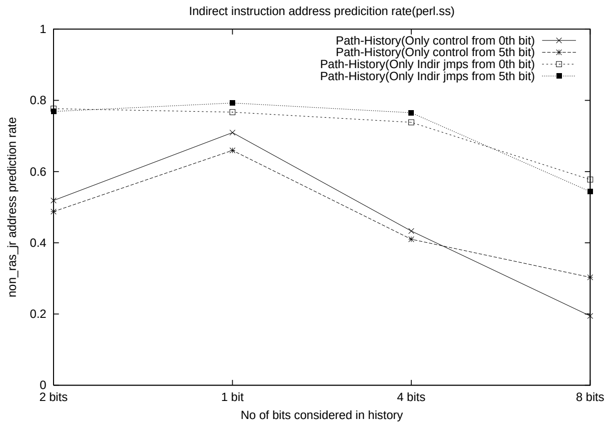


Figure 5: Target cache prediction accuracy for perl.ss

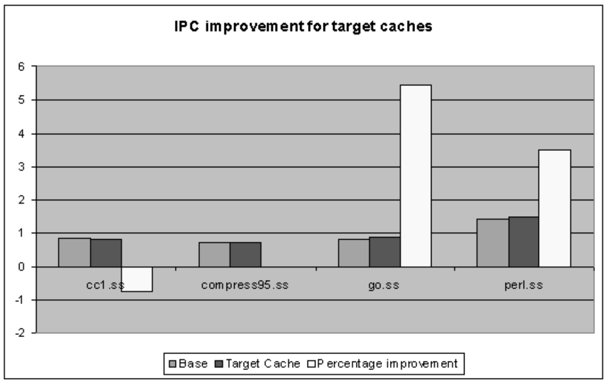


Figure 6: IPC improvements for target caches

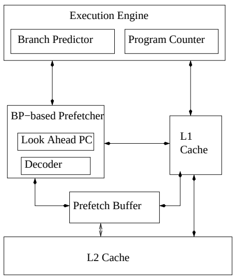


Figure 7: Branch Prediction based instruction prefetching scheme

predictors have a resonably high accuracy of prediction as show in section 4.1.3, the instructions needed in the future can also be predicted accurately and prefetched on advance. The branch predictors can be shared by the fetch as well as the prefetch stream. Such a prefetching scheme was introduced in [CHEN 97] and is depicted in figure . Branch prediction based prefetching achieves high performance by speculatively running ahead of the execution unit.

#### 4.2.1 Description

The branch prediction based prefetching scheme is similar to the Look Ahead stride prefetching scheme described in [BAER 95] with some features added from [CHEN 97]. The prefetching unit is an autonomous unit which runs down the instruction stream as fast as possible and brings all the instructions encountered along the path. When a branch instruction is encountered, the prefetching unit predicts the likely execution path using the branch predictor, records the prediction in a log, and continues. When the execution unit resolves a branch, it checks the log and resets the program counter of the prefetching unit when an error is found.

The additional hardware needed is as follows.

1. A look-ahead program counter (LA-PC) that is used in execution stream prediction. It is modified via the Branch prediction table.
2. An Outstanding Request List (ORL) that holds the addresses of in-progress or outstanding prefetches.
3. A Prefetch buffer (PB) that temporarily holds the prefetched instructions before they are moved into the instruction cache when an I-Cache miss ocuurs.
4. A FIFO prediction log (PL) which contains the adddress of predicted addresses and used to validate the predictions.
5. Prefetch Request Queue (PRQ) which contains the prefetch requests

Initially both the actual PC and the look ahead PC contain the same address. Every cycle, the look ahead PC checks the branch target buffer(if it is not used by the instruction fetch unit in the same cycle). If the address is found in the BTB, then the prefetching unit assumes that this is a branch and makes an entry in the log and the look ahead PC is changed to the target. So the prefetching unit will examine the instructions(via branch predictors) at the new look ahead PC from the next cycle. Simultaneously the prefetching unit checks the top of the prefetch request queue and checks if the prefetch address is present in cache. This probing is done using a tag cache, which stores only the tags of the instruction cache and is kept in sync with the i-cache. So a separate read port is not required for this probing. If the top of the prefetch request queue is not in the cache, then a prefetch is initiated, if the cache is available.The prefetching unit runs ahead of fetch unit whenever an i-Cache miss occurs and the fetch unit is stalled.

When the execution unit resolves a branch, the actual outcome is compared with the one predicted by the prefetching unit, which is recorded in the log. If the actual outcome matches the one predicted, the item is removed from the log. However, if the acutal outcome differs from the one predicted, the prefetching unit has gone down a wrong path, then the entire

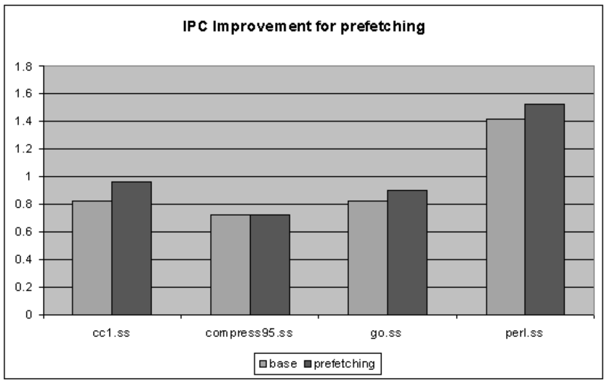


Figure 8: IPC improvement for prefetching

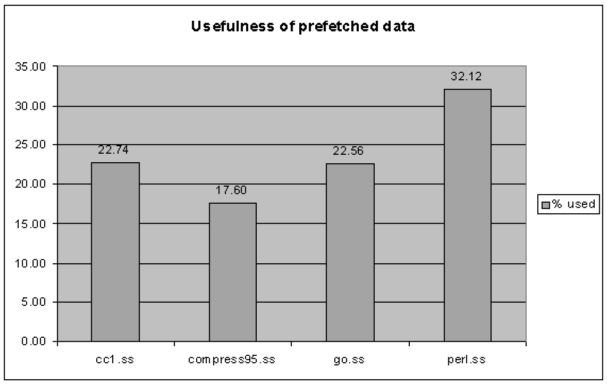


Figure 9: Usefulness of prefetched data

PL and PRQ are flushed and LA-PC is reset to the actual PC and the prefetching unit goes on. Thus size of the log controls how far ahead can the look ahead PC go.

#### 4.2.2 Performance enhancements

In our experiments, we used a prefetch buffer of about 32 entries, and the other parameters did not really affect the performance. The results are summarized in figure 8. We see an improvement for all th e benchmarks except compress which has a zero miss rate because of a small memory foot print. It is interesting to see certain charactersitics like usefulness. The usefulness of the prefetched instructions is summarized in figure 9. The figure shows the percentage of the total prefetched instructions that are actually used i.e., they miss in L1 cache but are found in prefetch buffers.

### 4.3 Bi-mode predictor

There are some additions that we added did not show any improvements at all. One such feature is the bimode predictor [CLEE 97]. A branch miss prediction can occur in the following two cases. A branch which was taken for the last n times changes behaviour and changes to not-taken (eg. end of loop). The other case occurs when the prediction states in the BPT is corrupted due to destructive aliasing. The first case cannot be handled easily whereas the destructive aliasing can be reduced. The bi-mode predictor tries to replace destructive aliasing with neutral aliasing.

#### 4.3.1 Description

In the Bi-mode predictor scheme, the BPT is split into even parts. One of the parts is the choice BPT, which is just a bimodal predictor. The other two parts are direction BPTs, one is a "taken" direction BPT and the other is a "not taken" direction BPT. The direction BPTs are indexed by the branch address xored with the global history. When a branch is present, its address points to the choice BPT, which in turn chooses between the "taken" BPT and the "not taken" BPT. The prediction of the direction BPT choosen by the choice BPT serves as the prediction.

As a result of this scheme, branches which are biased to be taken will have their predictions in the "taken" direction BPT, and branches which are biased to be not taken will have their predictions in the "not taken" direction BPT. This ensures that at any given time most of the information stored in the taken direction BPT entries is taken and any aliasing is more likely not to be destructive. The same phenomenon happens in the not taken direction BPT. The choice BPT serves to dynamically choose the branches' biases.

#### 4.3.2 Impact on performance

With the bimode predictor we could not get any improvements. In fact the performance decreased by about 5-6% for most of the configurations.

### 4.4 Data prefetching

The goal of any prefetching scheme is to reduce the stall time of the processor by bringing in the data into the cache ahead of its actual use so that it can be accessed without delay. Prefetching of data should be initiated at the appropriate time so that the data is available when required. Prefetching the data too early will result in cache pollution by replacing useful data. On the other hand, prefetching too late will result in data not being available when required. Another factor that limits the performance of prefetching is its accuracy, which is the fraction of prefetched data that will be used in the future. This implies that the effectiveness of the data prefetching scheme is evaluated on the accuracy and timeliness of prefetched data. Prefetch requests result in additional memory traffic. This should also be taken care of.

A survey of data prefetching techniques can be found at [VAN 00]. There are several kinds of prefetching like next n-line prefetching, stream buffers, stride prefetching etc., All the methods try to exploit some locality in the data access pattern of the program. There are also other access patterns like pointers which do not have straightforward temporal or spatial locality. Since gcc, perl and go have a lot of linked data structures we explored content based data prefetching schemes [AMIR 98]. A content based prefetching scheme keeps track of the links between a load that reads a value from memory and another load that uses this value. In such cases, if the distance between the producer load and the consumer load is greater the miss latency of level 1 cache, then prefetch could be initiated when seeing the producer load. However in our case it turned to be of not much help because the distance between producers and consumers was lesser than the L1 miss latency and also most of such data hit in cache. We inferred this by profiling the benchmarks. The graph has been omitted due to space considerations. So we resorted to a simpler form of prefetching the stride prefetching.

#### 4.4.1 Description

The basic idea of the stride prefetching scheme is to keep track of data access patterns in the form of an effective address and stride. The stride prefetching scheme that was implemented was based on the architecture introduced in [BAER 95]. Unlike the conventional cache prefetching, which associate a successor block to the block being currently referenced, in this scheme the prefetching is based on the prediction of the instruction stream. This implies that the architecture should contain a branch prediction table (BPT) which is already present in the base configuration. The additional hardware needed in this scheme are as follows:

1. A look-ahead program counter (LA-PC) that is used in execution stream prediction. It is modified via the Branch prediction table.
2. A Reference Prediction Table (RPT) that is used for associating load PC with effective address and stride
3. An adder used for stride computation
4. An Outstanding Request List (ORL) that holds the addresses of in-progress or outstanding prefetches.

An RPT entry corresponds to a load/store instruction and contains the last addresses seen by that instruction and the current predicted stride. It also contains a two bit state information which basically contains the past history. When the LA-PC hits a load/store instruction, an RPT lookup is performed. If the state indicates that a prefetch can be made and the data to be prefetched is not already in cache or in ORL, then the load address is added to the ORL. If the LA-PC hits a branch instruction, a prediction is made using the BPT. The probability that the predicted instruction stream as seen by the LA-PC is incorrect increases with the distance between the actual PC and LA-PC. Hence a Look Ahead limit equal to 1.5 times [BAER 95] the miss latency of level

| Cache | Size | BSize | # Sets | Assoc | Tags |
|---|---|---|---|---|---|
| L2 | 128K | 128 | 256 | 4 | |
| DL1 | 8K | 32 | 32 | 8 | 704 |
| IL1 | 16K | 128 | 4 | 32 | 368 |
| Total | **24K** | | | | **1072** |

Table 7: Cache Configurations

1 cache is imposed. When the LA-PC is reset due to an incorrect branch prediction, the ORL is flushed

#### 4.4.2 Impact on performance

Unfortunately data prefetching resulted in a very minimal improvement in the data cache miss rates. On further investigation of the reason for this behaviour the following was observed. When the RPT presents a possible target for prefetch it is checked whether the data is already in cache. It was found that a very high percentage of these load addresses was already a cache hit. To overridde this effect the Look ahead limit was increased. A prefetch buffer was also added to temporarily store the data to prevent cache pollution effects. Though this resulted in prefetching more data it turned out that the prefetching was inaccurate (prefetching proceeded in a wrong execution path) and the data in the prefetch buffer turned out to be useless.

This led to the conclusion that for the benchmarks and the level 1 data cache sizes that were considered, the improvement seen by incorporating data prefetching is very minimal.

# 5 Final Results

With our experiments, we used the **target cache** and **instruction prefetching** in our enhanced hardware. The final distribution of memory is given in Tables 7, 8, 9 and

The overall enhancement in the IPCs is indicated in figure 10. The third bar for each of the benchmarks show the overall percentage improvement in IPC for the enhancements.

| Component | Configuration | Size |
|---|---|---|
| Meta table | 1024 | 256 |
| Bimode | 2048 | 512 |
| Two level | L1-1 (global history) | 13 bits |
| | L2 - 8192 | 2048 |
| | | **2818** |

Table 8: Branch Predictor Configuration

| Number of sets | 32 |
|---|---|
| Associativity | 8 |
| Size | 1 K |
| Tags | 864 bytes |

Table 9: BTB Configuration

| Feature | Memory |
|---|---|
| Target Cache | 2K |
| Inst. Prefetcher- Tag Cache | 368 |
| Inst. Prefetcher - ORL | 64 |

Table 10: Memory requirements of enhancement features

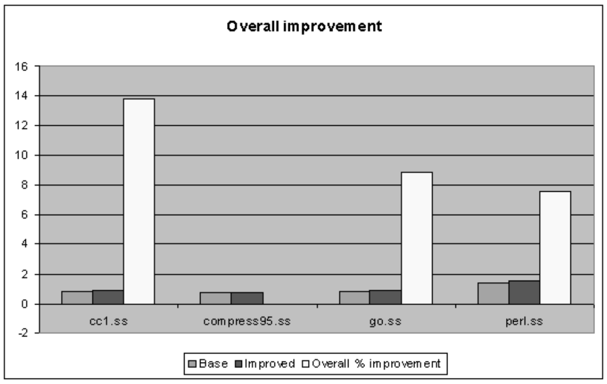


Figure 10: Over IPC enhancement

## References


[BAER 95] J.L. Baer, and J.L. Chen, "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Transactions on Computers, Vol. 44, No. 5, pages 609-623, May 1995.

[CHAN 97] P.Y. Chang, E. Hao, and Y. Patt, "Target Prediction for Indirect Jumps", Proceedings of the International Symposium on Computer Architecture, pages 274-283, June 1997.

[CLEE 97] C.-C. Lee, I-C.K. Chen, and T. Mudge, "The Bi-Mode branch Predictor", Proceedings of the 30th Annual International Symposium on Microarchitecture, November 1997.

[CHEN 97] I-C.K. Chen, C.-C. Lee, and T. Mudge, "Instruction Prefetching Using Branch Prediction Information", Proceedings of International Conference on Computer Design, pages 593-601, October 1997.

[EDEN 98] A. Eden and T. Mudge, "The YAGS Branch Prediction Scheme," 31st Annual International Symposium on Microarchitecture, pages 69-77, Dec. 1998.

[VAN 00] Steven P. Vanderwiel and David J. Lilja, "Data prefetch mechanisms",ACM Computing Surveys, vol 32, No. 2, pages 174-199, 2000

[NAIR 95] Ravi Nair, "Dynamic path-based branch correlation". In proceedings of the 28th Annual ACM/IIEEE International symbosium on Microarchitecture, pages 15-23, 1995.

[AMIR 98] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi, "Dependence based prefetching for linked data structures", Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, SanJose, California, October 1998.