**A Study of Mispredicted Branches Dependent on Load Misses in**

# Continual Flow Pipelines

Pradheep Elango, Saisuresh Krishnakumaran, Ramanathan Palaniappan

{pradheep, ksai, ram}@cs.wisc.edu

**Dec 2004**

**University of Wisconsin - Madison**

**Abstract**

*Large instruction window processors can achieve high performance by supplying more instructions during long latency load misses, thus effectively hiding these latencies. Continual Flow Pipeline (CFP) architectures provide high-performance by effectively increasing the number of actively executing instructions without increasing the size of the cycle-critical structures. A CFP consists of a Slice Processing Unit which stores missed loads and their forward slice inside a Slice Data Buffer. This makes it possible to open up the resources occupied by these idle instructions to new instructions. In this project, we have designed and implemented CFP on top of Simplescalar. Further, we have compared conventional pipelines to CFPs by running them on various benchmarks in SPEC integer benchmarks suite. We also studied the behavior of mispredicted branches dependent on load misses, which turn out to be the main bottleneck in CFPs. We also compare the performance of CFPs with ideal and non-ideal fetch mechanisms.*

## 1 Introduction

### 1.1 Importance of Single Thread Performance:

Current trends indicate that chip multiprocessors are going to take over the industry. In order to economize the design, these multiple cores on single chips target both the mobile laptop industry as well as the high end servers. While the predominant goal of the mobile computers is to maximize single thread performance, high end servers consider higher throughput to be more important. Thus, improving single thread performance is vital.

### 1.2 Memory Bottleneck

The ever-growing gap between processor speed and memory speed makes the task of improving single thread performance very challenging. With every technological stride in processor speed, the speed difference becomes more blatant, and caches become increasingly important. Cache misses have become more and more expensive, as a result of the steady growth in the speed difference. While multiple cache levels reduce this miss penalty to a great extent, long latency cache misses are still very expensive.

For example, consider the situation immediately after a load miss at the L2 cache. The load miss occupies register file resources and scheduler resources as long as it is being serviced. And the vast speed difference means that typically this is going to take about 400 CPU clock cycles. Moreover, the direct and indirect dependents of this missed load instruction, (also called the *forward slice* of the load) soon occupy these cycle critical structures. Owing to the limited size of these structures, further instructions are blocked and stall the processor until the miss is serviced. In a conventional processor, the missed load will not let the other completed instructions behind it to commit, thus delaying execution by a lot of processor cycles.

### 1.3 Large Window Size

Traditional solutions to the memory bottleneck problem have involved increasing the instruction window size. In order to hide the glaring load latency, the processor needs to have a lot of actively executing instructions in its pipeline. This can be done by increasing the instruction window size. However, naively increasing instruction window size also increases processor cycle time, since it is one of the cycle critical structures. Apart from this, large instruction window processors are also power-hungry, and consume a lot of chip space. In the light of the growing multiprocessor industry, these negative points will only increase.

### 1.4 Continual Flow Pipelines

A CFP effectively increases the size of the instruction window, without increasing the size of the cycle critical structures, thereby providing a power-efficient solution. The main problem with traditional pipelines is the idle occupancy of the critical structures by the load misses and their forward slices. CFPs overcome this by releasing these instructions from the main pipeline. Instructions are moved out of the

pipeline, and are subsequently stored in a secondary buffer. The processor is likely to find instructions that are totally independent of the load miss and its forward slice. Therefore, this release of resources is expected to produce a marked improvement in processor performance.

A Continual Flow Pipeline consists of a Slice Processing Unit, apart from the main pipeline. The Slice Processing Unit performs three important functions:

1. Releasing the resources occupied by load misses and their forward slices, by transferring them to the Slice Data Buffer.
2. Ensuring no data dependence and memory dependence violations occur,
3. Ensuring the correct execution of instructions that are independent of the load misses and their forward slices.
4. Re-inserting the load misses and their forward slices back into the main pipeline as soon as the miss is serviced.

### 1.5 Organization of the Report

Section 2 provides an overview of the CFP architecture, and discusses the finer details. Section 3 discusses our implementation of CFP in Simplescalar. We present our results in the Section 4, and discuss related work in Section 5. Section 6 concludes the report.

## 2 CFP Architecture

The Slice Processing Unit (SPU) is the most important unit in a CFP. The Slice Processing Unit consists of the following components:

- Slice Data Buffer
- Rename Filter
- Remapper

An overview of the CFP architecture is illustrated in Fig 1.

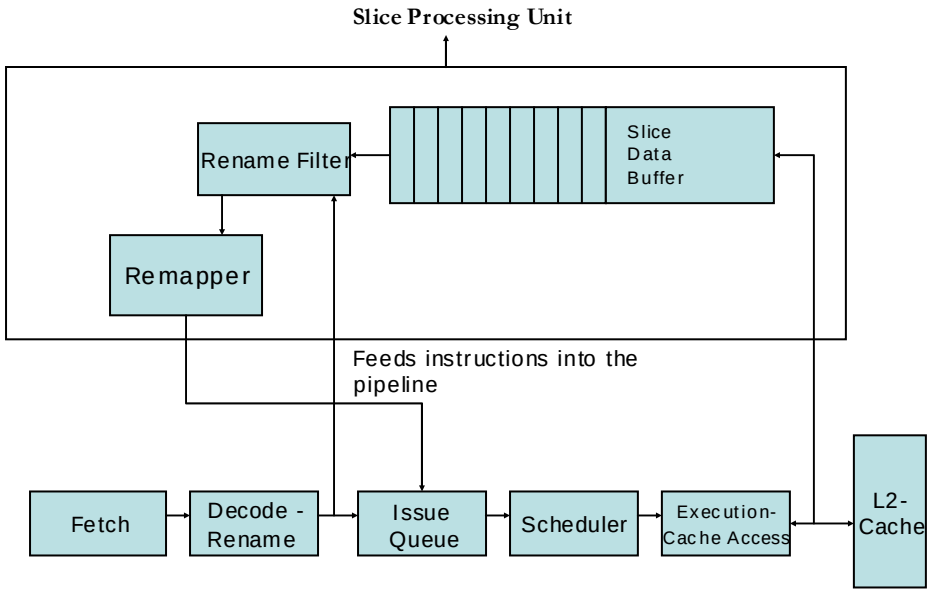


**Fig. 1: CFP Architecture**

***Slice Data Buffer (SDB)***: Loads that miss in the L2-cache and its dependent instructions are buffered in the Slice Data Buffer which is a FIFO. The SDB can contain multiple slices corresponding to independent load misses. This structure is not cycle-critical and can be of a bigger size than the instruction window because it is not accessed every cycle. The SPU transfers all load misses and their forward slices into the SDB. In order to achieve this, the Slice Processing Unit considers the dependent operands of the slice instructions as ready before inserting into the slice. All the ready operands of an instruction are read before it is put into the SDB.

***Rename Filter***: The rename filter has an entry for each logical register and contains the id of the instruction that last wrote the register. When an instruction is renamed in the decode/rename stage, the rename filter is updated. When an instruction is reinserted into the pipeline, it compares its own identifier with the instruction identifier in the rename filter entry corresponding to its destination register. If the identifiers do not match, then the instruction is prevented from writing the register. In short, the rename filter prevents WAW hazards.

***Remapper:*** Instructions release all their resources (registers, scheduler entries) while entering the slice. Hence when instructions are reinserted into the pipeline, the remapper allocates new scheduler entries and physical registers to these instructions.

When a load miss occurs in a conventional superscalar processor, the load stays in the pipeline occupying scheduler entries and registers, thereby, preventing independent instructions from entering the pipeline. While the load is in the SDB, its forward slice enters the SDB, releasing cycle critical resources like registers and scheduler entries as shown in Fig 2a and b. Instructions that do not miss and do not depend on any load misses directly or indirectly, enter and exit the pipeline very quickly as their operands are ready. This effectively increases the number of actively executing instructions in the pipeline.

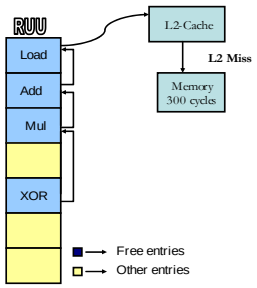


**Fig 2a: A load miss and its forward slice stalling a conventional pipeline.**

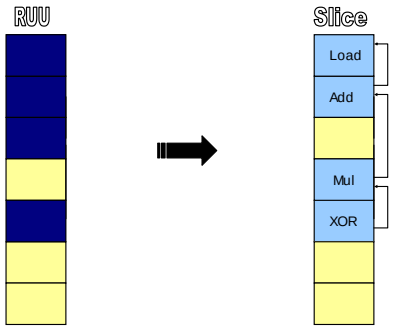


**Fig 2b: Load Miss and its forward slice moved to SDB freeing the main RUU for other independent instructions.** *Dark Blue* indicates free entries. *Light Blue* indicates forward slice in Slice. *Yellow* indicates other entries.

## 3. CFP Implementation

Implementation of CFP in Simple Scalar required major modifications to the sim-outorder core [8]. Several new data structures and modules were integrated into the simple scalar framework. A brief explanation of the data structures and the modules is given below.

### 3.1 New Data Structures:

#### 3.1.1 RUU_OOO

Removing instructions from the RUU creates "holes" in the RUU of the original Simplescalar implementation since it is designed as an array. This opens up two issues to be handled, namely, allocation of a new entry in the RUU, and maintaining order of instructions for commit. Allocation of a new entry can be handled by maintaining a "free" bit in every RUU entry which indicates if the entry is occupied or not.

Maintaining correct order of instructions is, however, trickier. In order to overcome this issue with minimal code modifications, we have designed a doubly linked list implementation for the RUU.

```
struct RUU_OOO {
        struct RUU_station *rptr;
        struct RUU_OOO *next, *prev;
};
```

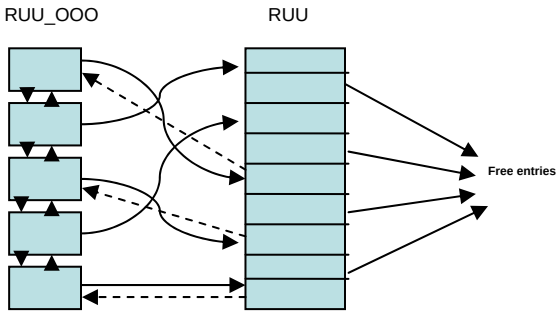


**Fig 3: RUU_OOO entries point to entries in RUU which correspond to active instructions. solid arrows from RUU_OOO to RUU are the *rptr's* in the structure. Dashed arrows from RUU to RUU_OOO represent the o*ptr*'s.**

Since instructions can enter and leave the RUU at any arbitrary time, the RUU can no longer be considered to be a queue with all the free entries compacted at one end. With this model, free entries can be present anywhere in the RUU. Hence we have effectively created an RUU station (a doubly linked list) which keeps track of all the current instructions in the RUU. The field *rptr* points to an RUU entry while the other fields are used to traverse the linked list in any order. Similarly a field by name 'optr' is present in each RUU entry, which points to its corresponding RUU_OOO node. Thus the RUU can be manipulated as a queue with the help of the RUU_OOO. In short, RUU_OOO serves as an index into the main RUU.

### 3.1.2 Slice Processing Unit

The Slice Processing Unit consists of two queues, a slice_LSQ to hold the loads and stores and a slice_RUU to hold other instructions.

```
struct Slice
{
        struct RUU_station *cfp_slice_RUU;
```

struct RUU_station *cfp_slice_LSQ;

}

## 3.2 Data Structures Modified:

### 3.2.1 RUU

New entries should be added to RUU to support out of order commit, to handle complexities involving the movement of instructions to and from the slice, and to maintain pointers to corresponding entries in the RUU_OOO. The following fields have been added to the struct RUU_station.

1. free bit: This bit informs whether an RUU entry is free or not.
2. later_into_slice bit: This bit indicates whether an instruction should get into the slice at a later point of time after reading its non-poisoned operands. Poisoned operands are operands that are produced by instructions in slice.
3. inames – stores the input register names
4. now_in_slice – indicates whether the instruction is in the slice or not.
5. is_dummy – this bit is used when the RUU becomes full while instructions are being re-inserted into the pipeline. ( This is explained in greater detail below )
6. optr – points to its corresponding RUU_OOO

### 3.2.2 CV_link (register status unit)

*CV_link* maintains the status of each register. It includes the following additional fields.

1. poison – indicates whether the instruction producing this register is in the slice or not.
2. seq – sequence number of the above mentioned instruction

## 3.3 Insertion of an instruction into the Slice

An instruction can enter into the slice at three different stages

1. ruu_issue ( )
2. ruu_dispatch ( )
3. ruu_writeback ( )

**ruu_issue:**

When a load miss is encountered in the ruu_issue stage, the module *insert_into_slice* is invoked which inserts the load and all its dependent instructions (forward slice) into the slice. The poison bit of the load's destination register is set and all its dependent instructions inherit this recursively.

**ruu_dispatch:**

During dispatch, lot of case analysis needs to be done in deciding whether an instruction should get into the slice or not.

| Serial No | Instruction Type | # Ready Operands + | #Not Ready Operands * | # Poisoned Operands δ | Later into slice | Action Taken |
|---|---|---|---|---|---|---|
| 0 | ALU | 3 | 0 | 0 | No | No |
| 1 | ALU | 2 | 1 | 0 | No | No |
| 2 | ALU | 2 | 0 | 1 | No | Yes |
| 3.a | ALU | 1 | 1 | 1 | Yes | No |
| 3.b | ALU | 0 | 2 | 1 | Yes | No |
| 3.c | ALU | 0 | 1 | 2 | Yes | No |
| 4 | ALU | 1 | 2 | 0 | No | No |
| 5 | ALU | 1 | 0 | 2 | No | Yes |
| 6 | ALU | 0 | 3 | 0 | No | No |
| 7 | ALU | 0 | 0 | 3 | No | Yes |

**Table 1: Table that indicates the action to be taken for every non-memory instruction that enters the pipeline. A yes in the "action taken" column indicates that the instruction is to inserted into the slice, and a "no" indicates it will not be inserted into slice**

* Not Ready – Operands that are currently unavailable but produced by an instruction in the main RUU.

+ Ready – Operands which are readily available.

δ Poisoned – Operands that are produced by an instruction in the slice.

Cases 3.a, 3.b and 3.c are non-trivial in which an instruction does not get into the slice even though one of its operands is poisoned. In CFP, an instruction should read all of its not ready operands before getting into the slice. Hence, an instruction with poisoned operands (whose other operands are not ready) has its *later_into_slice* bit set. This bit will be used to check whether this instruction should be inserted into the slice or not in the ruu_writeback() stage when the not ready operands become available.

A similar analysis is done for loads and stores. Load has only one input operand while a store has two input operands.

| Serial No | Instruction Type | #Ready Operands | # Not Ready Operands | #Poisoned Operands | Later_into_slice bit set ? | Action Taken |
|---|---|---|---|---|---|---|
| 0 | Load | 1 | 0 | 0 | No | No |
| 1 | Load | 0 | 0 | 1 | No | Yes |
| 2 | Load | 0 | 1 | 0 | No | No |
| 3 | Stores | 2 | 0 | 0 | No | No |
| 4 | Stores | 0 | 2 | 0 | No | No |
| 5 | Stores | 0 | 0 | 2 | No | Yes |
| 6 | Stores | 0 | 1 | 1 | Yes | No |
| 7 | Stores | 1 | 0 | 1 | No | Yes |

**Table 2: Table that indicates the action to be taken for every memory instruction that enters the pipeline.**

*ruu_writeback ():*

In the ruu_writeback () stage, completed instructions feed their values to dependent instructions and set their corresponding *ready* bits. Instructions that have their later_into_slice bit set (has one or more poisoned operands), are inserted into slice when all of their non poisoned operands become ready.

### 3.4 Reinsertion of instructions from the slice to the core pipeline

The reinsert_into_pipeline () method is invoked whenever a load returns from a miss. This method inserts the load and all its dependent instructions back into the RUU. While reinserting, new *RUU* and *RUU_OOO* entries are allocated for these instructions. During this process the RUU can become full, during which a *dummy* entry is added to the *event_queue* to try reinserting the remaining instructions in the next clock cycle. The *dummy* entry is just a NOP, (with its *is_ dummy* bit set) whose output dependency list points to the instructions in the slice that should be reinserted into the pipeline.

**Handling Full RUU**

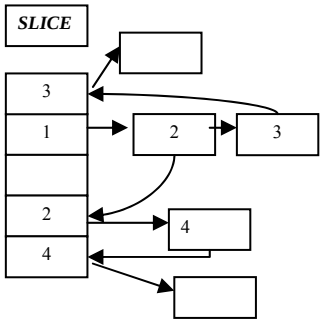


*Fig. 4* Arrows represent the dependences between instructions in the slice. Each slice entry has a list of pointers to instructions consuming its operands.

In the above figure (Fig. 4), let's assume entry 1 to be a load which is in the slice due to a L2 cache miss. When the load returns from the miss, its forward slice (in this case 2, 3 and 4) will be reinserted into the pipeline along with the load. Consider a case where 1 and 2 have been reinserted into the pipeline, after

which the RUU becomes full. We handle this case as shown below in Fig.5. Two dummy entries are created in the event queue whose output dependency lists point to nodes 3 and 4 respectively. These dummy entries are given a latency of 1 clock cycle so that they will be popped off in the next cycle. If any RUU entries become available in the next cycle, then 3 and 4 will be inserted into the RUU else the dummy entries will be retained in the event queue until 3 and 4 get into the RUU.

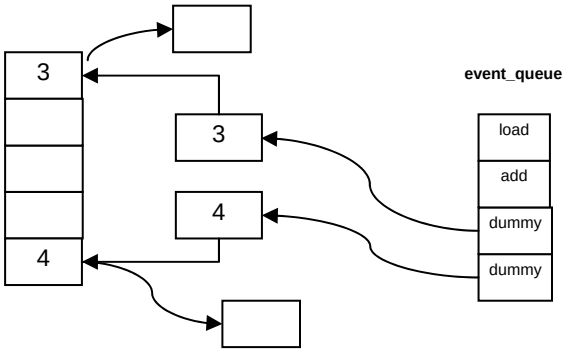


***Fig. 5:*** Dummy entries are inserted into the event queue whose output dependence lists point to instructions 3 and 4 respectively.

### 3.5 Memory Dependences

A straightforward solution to the memory dependence problem between instructions in the main pipeline and those in the slice is as follows:

- A conflict of an incoming load or store with a load or store in the slice results in the insertion of the corresponding memory instruction into the slice.
- A memory instruction with an unresolved address is also treated in the same way as a conflict.
- The instructions inserted above are re-fed into the pipeline along with their counterparts.

This is a very simple solution that lacks efficiency.

A more efficient way to handle this, at the cost of additional complexity, would be as follows. Whenever the processor comes across a load that has a may-be dependence, it adds the load address, the instruction sequence number, the old load value at this address (tricky) and the pipeline continues execution with this address. Whenever a store instruction that was in the slice completes later, this address is compared with the load addresses in the table. If it matches, then the value that was used for load was incorrect, and hence all instructions following the load will be squashed. This approach is similar to the ALAT approach used by the Itanium architecture [7].

### 3.6 Instruction Commit

We have used *CPR (Checkpoint Processing and Recovery)* [2] technique to commit the instructions. Hence instructions can commit out of order but the system state was checkpointed whenever a branch was encountered. In the event of a mispredict, the system state was restored to the corresponding checkpoint. The RUU_OOO station simplifies out-of-order commit, since it effectively manages the RUU as a contiguous unit irrespective of how instructions enter or leave the RUU.

## 4. Simulation Results:

We ran the simulation for 500M instructions, fast forwarding 100M instructions for all the benchmarks. We have used the following system configuration.

| System Parameter | Value |
|---|---|
| L1 I-cache | 512:32:1:l |
| L1 D-cache | 128:64:4:l |
| Unified L2-cache | 8192:64:4:l |
| L1 I-cache access latency | 1 cycle |
| L1 D-cache access latency | 1 cycle |
| L2-cache access latency | 8 cycles |
| Memory Access Latency (L2-miss penalty) | 256 cycles |
| # Memory Ports | 32 |
| RUU size | 128 |
| LSQ size | 64 |
| SDB size | 1024 |

**Table 3: System Parameters for Simulation**

### 4.1 CFP vs. Large Instruction Window Processor

It is interesting to note the relative performance of CFP with conventional pipelines that have large instruction windows. Our simulation results show that CFP performs only slightly better than the conventional pipelines with large instruction windows.

| | |
|---|---|
| Continual Flow Pipelines | 0.6614 |
| Conventional Processors with Large Instruction Windows | 0.6693 |

**Table 4: CPI Comparison –CFP vs. Large Window Machines**

This can be attributed to the following reasons:

1. In CFP, instructions in the forward slice read their ready operands before entering the slice. Whereas in conventional machines, these entries hold register resources until all their input operands become ready.
2. The performance difference is not very significant because of additional overhead involved in moving instructions to and from the slice.
3. Moreover, increased complexity involved in resolving the dependences within a large instruction window will increase the clock cycle time.

### 4.2 Overlap of Load Misses in CFP

The graph in Fig 6 shows the arrival and departure times of load misses for CFP and conventional pipelines for the gcc benchmark. Instructions arrive much earlier in CFP compared to conventional pipelines as CFP is able to overlap long latency load misses. Let us take a deeper look at this graph by zooming in on the left most part of this graph

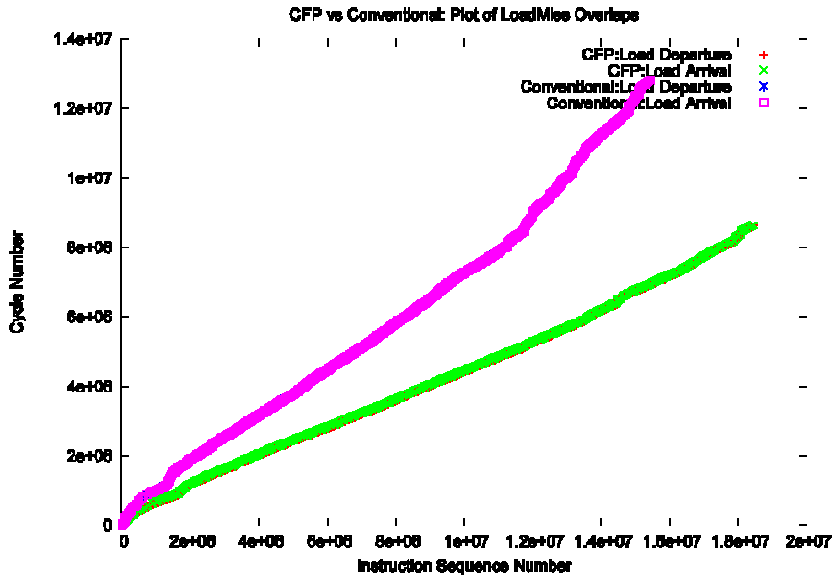


*Fig.6.* Instructions arrive much earlier in CFP compared to conventional pipelines as CFP is able to overlap long latency load misses.

From the graph in Fig.7, we can observe that the plot of the conventional machine rises more steeply than that of CFP's. The gradient difference is very obvious when we have clustered loads misses at points A & B in the graph. This is because CFP exposes the cache misses early but conventional machines suffer from structural hazards.

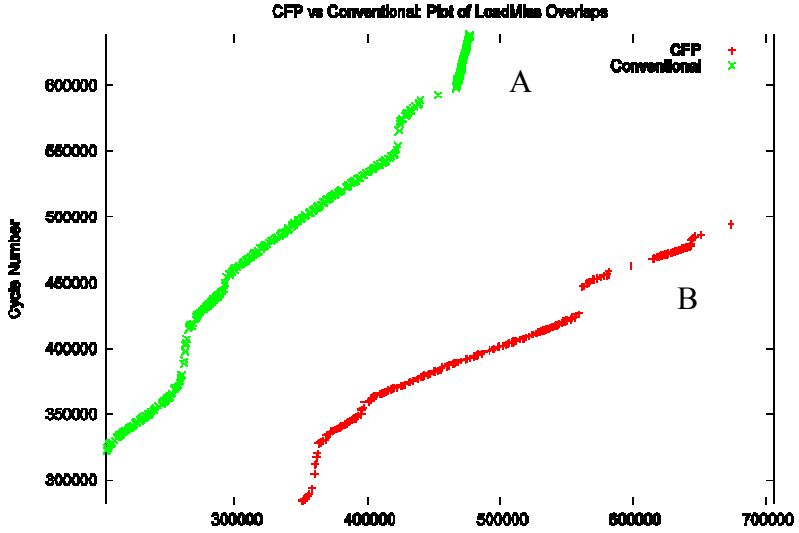


*Fig.7.* Graph showing the gradient difference of load departures between a conventional pipeline and CFP. The steeper gradient in the conventional plot indicates load misses depart much later than load misses in CFP, which highlights the CFP's ability to overlap misses, and to expose cache misses early.

Graph in Fig.8 shows the overlapping of load misses in CFP. This is the region where the clustered load misses benefit from CFP's overlapping effect.

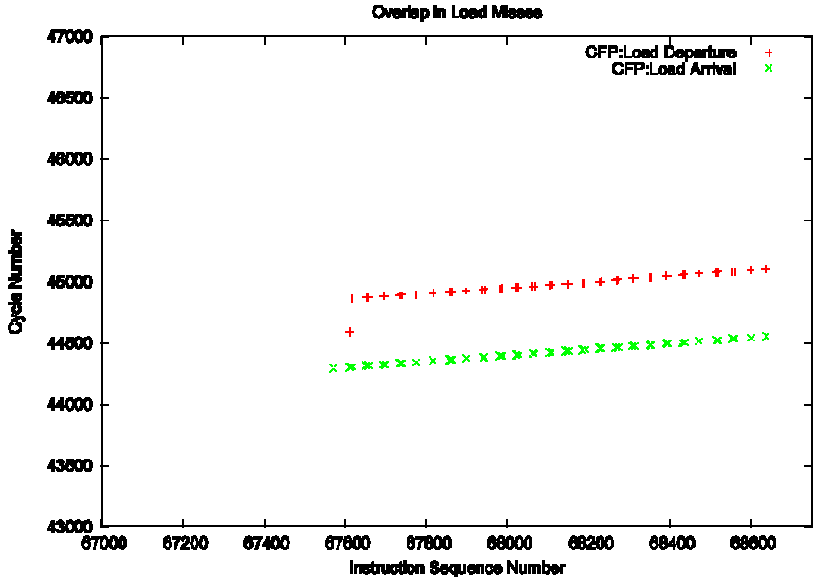


*Fig.8.* Graph exposing clearly the overlap in load misses by a CFP. The x-axis indicates the instruction sequence number of a load miss. CFPs expose cache misses early and service a lot of load misses in parallel.

Fig.9 shows the corresponding load misses in the case of a conventional machine. Here, loads enter into the pipeline only after the previous one has been serviced because the long latency load misses and their dependant instructions hold the RUU entries until the load is serviced. In CFP the load missed instruction is moved into the slice and hence new instructions come into the pipeline enabling multiple load misses to be serviced in parallel.

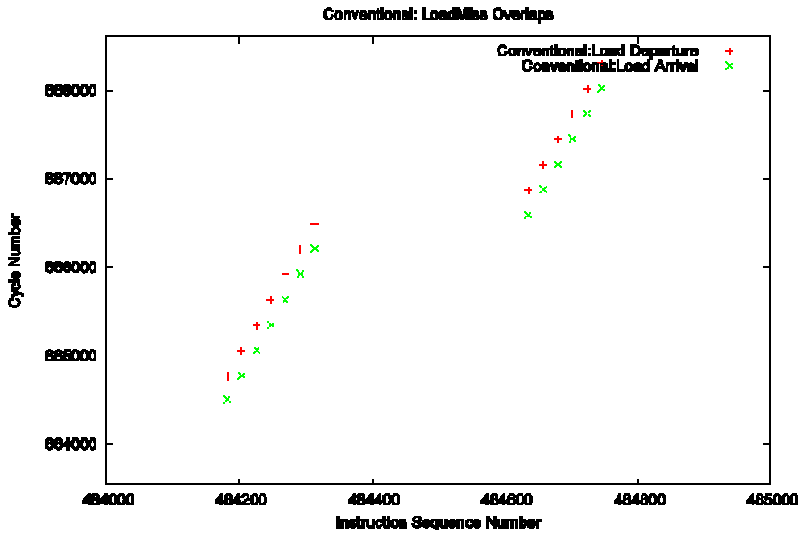


*Fig.9.* Graph displaying the arrivals and departures of load misses corresponding to the load misses plotted in Fig 8. The steeper slope indicates the delay in exposing cache misses.

## 4.3 Performance of CFP vs. Conventional Machines

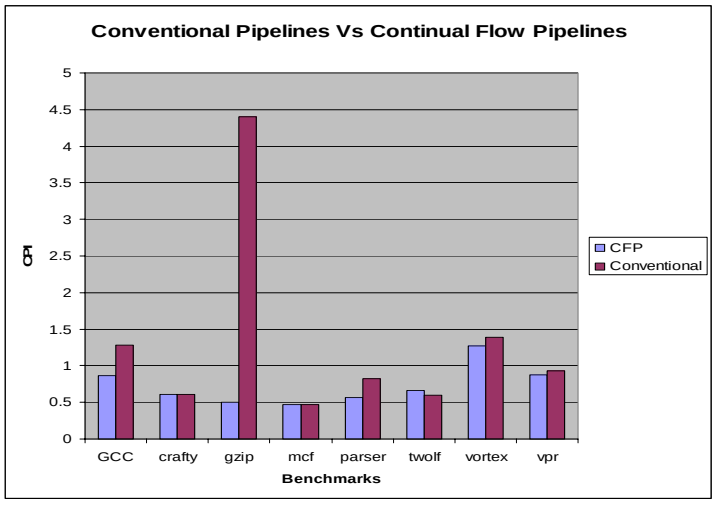


*Fig.10.* Graph comparing the CPI of CFP vs. Conventional Processors

The above graph compares the CPI of conventional machines with that of CFP based machines. CFP based machines outperform conventional machines with the exception of *twolf*. This could be attributed to one of the following factors:

- The total number of load misses which causes structural blockage in conventional machines might be lower in this case and hence CFP is not able to overlap the load misses.
- The ability to exploit independent instructions also affects the CFP performance.

## 4.4 Impact of Fetch Mechanism on CFP Performance

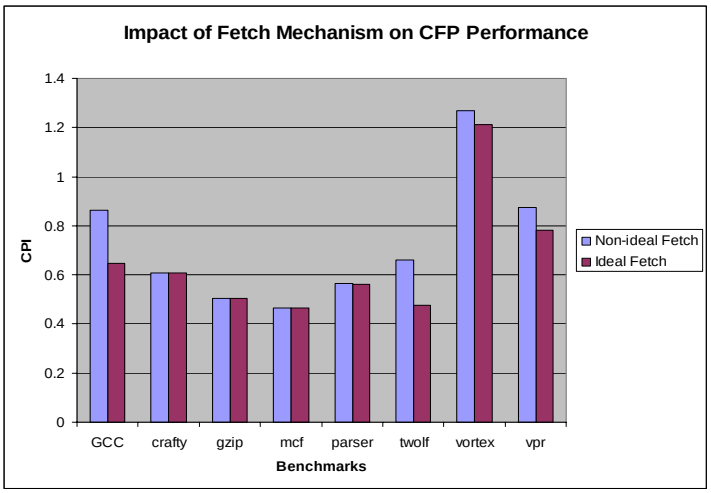


*Fig.11.* This graph shows that ideal fetch mechanism provides a better performance than a non-ideal fetch.

A good fetch mechanism could improve the performance of CFP. This can be attributed to two reasons. First, long latency instruction misses could prove to be very expensive. Second, since L2 cache is usually unified, instruction cache misses could contend with the data in L2 cache, thereby increasing the data misses in L2. Assuming an ideal cache, as in [1], therefore could lead to a better performance. We validated this with a few experiments, as can be seen in Fig 11. Some benchmarks do not have a better performance improvement.

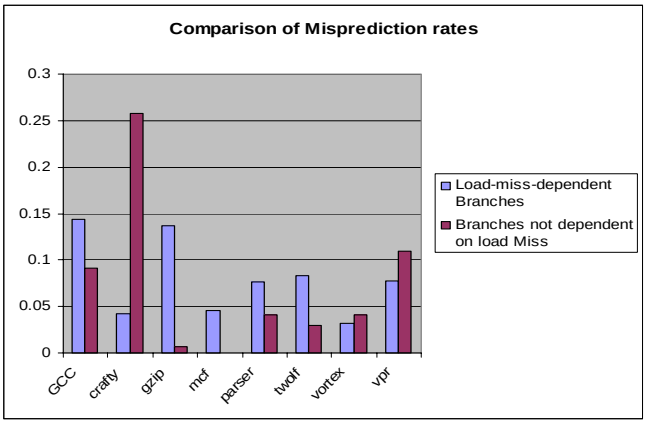


*Fig.12.* The graph shows that, generally, misprediction rates for branches dependent and independent on load misses are not the same.

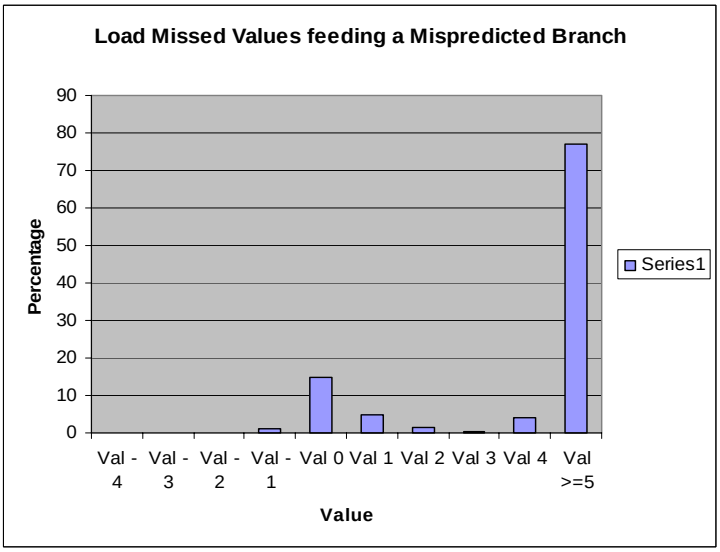


*Fig.13.* Distribution of values loaded by long latency load misses.

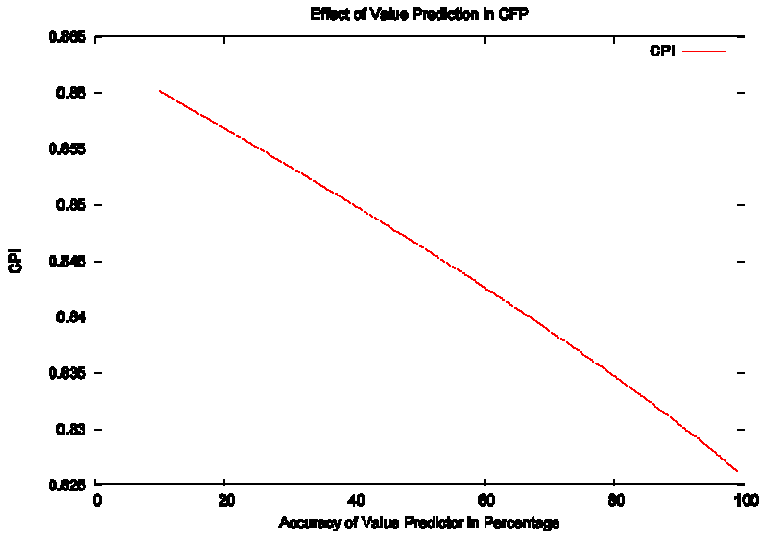


*Fig.14.* Graph demonstrating the effect of a value predictor with varying levels of accuracy on CPI.

The main bottleneck in CFP is the branch accuracy. Penalty associated with mispredicted branches dependent on load misses could be very high, because the branch will be resolved only when the load miss returns, and there could be a high number of instructions to be quelled when the branch is resolved. This turns out to be the main limiter of CFP performance [1]. Further, misprediction rates of branches dependent on a load miss do not seem to follow the same trends as the misprediction rates of branches independent on load miss, as can be seen in Fig. 12.

30% of load misses feed into mispredicted branches, according to [3]. Since this is a considerable fraction of misses, this requires more study. However, we took a sampling of the values that were loaded into the missed loads, which were subsequently feeding the branches that were mispredicted. Fig 13 shows the results. Not surprisingly, we found that the value 0 was the most commonly used value for comparison. There were a few other spikes, notably at values like -1 and 5. While 15% of the values were 0's, there was no comparable spike at any other value.

We reckoned that a good value predictor could augment the performance improvement provided by CFP because, a good value predictor can feed a correct value to a branch very early, thereby quickening the step where branches dependent on this load are resolved. This will lead to a reduction in the penalty associated with mispredicted branches dependent on load misses if the predicted load value turns out to be correct.

Subscribing to this theory, we simulated a value predictor. This was done by randomly setting the load cache latency to 1, depending on the accuracy of the value predictor. For example, a 90% accurate value predictor will predict the correct value for the load 90% of the time, which implies that the load cache latency will be just 1 clock cycle for 90% of the time. Since, it was simple to simulate with few modifications to the simulator, we have plotted our results in Fig 14.

## 5 Related Work

Much work has been done by the architects to hide long memory latencies that result from a L2-cache miss. Runahead [4] execution pre-fetches potential load misses in advance when a load miss is being serviced and hence indirectly avoids a L2 cache miss. However, this scheme would work only if the load misses are clustered together. Moreover, runahead throws away all the instructions that were executed in the shadow of a load miss. Another proposal, Waiting Instruction Buffer [5] involves buffering the loads and the load dependent instructions into a buffer. WIB targets a ROB style processor and hence the size of the buffer must be the size of the target instruction window

## 6 Conclusions and Future Work

We have implemented CFP within simple scalar and have found that in most of the cases, CFP is able to hide long memory latencies which are the primary bottlenecks in a conventional processor. CFP achieves this by overlapping the load misses and hence effectively reduces the load miss penalty. Our simulation results show that branch prediction is still a limiting factor to CFP's performance. In particular, we observed that mis-prediction rates for branches dependent on load misses do not follow the same trend as other branches. We also observed that using a perfect trace cache (as was used by the authors of the CFP paper) gives better results. The implementation of CFP in simple scalar involved a very detailed case analysis. Almost 60% - 70% of the sim-outorder core was modified to incorporate CFP. The ruu_commit stage required significant hacking as the conventional simple scalar implemented in-order commit, while CFP is built upon a CPR based machine.

## Acknowledgment

We would like to thank Professor Mark D. Hill for giving us the wonderful opportunity and motivating us to work on this research project. We would also like to thank him for his valuable guidance and encouragement that helped shape the project.

## References:

[1] S.T. Srinivasan, R.Rajwar, H.Akkary, A.Gandhi, M.Upton. Continual Flow Pipelines. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems,* October 2004.

[2] H. Akkary, R.Rajwar, and S.T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture,* December 2003.

[3] T.Karkhanis and J.E. Smith. A Day in the Life of a Data Cache Miss. In *Workshop on Memory Performance Issues,* 2002.

[4] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-Of-Order Processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture,* February 2003.

[5] A.R.Lebeck, J.Koppanalil, T.Li, J.Patwardhan and E.Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, May 2002.*

[6] Suleiman Sair and Mark Charney. IBM Research Report: Memory Behavior of the SPEC2000 Benchmark Suite. *IBM T.J. Watson Research Center.*

[7] C. McNairy and D.Soltis, "Itanium 2 Processor Microarchitecture," *IEEE Micro, vol. 20, Issue 5, Sept 2000, pp. 24-43.*

[8] http://www.simplescalar.com

[9] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach", *Morgran Kaufmann Publishers, Inc., San Mateo, CA 2002.*