International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 5, May-2014, pp: (256-264), Impact Factor: 1.252, Available online at: www.erpublications.com

Techniques to increase Memory Level Parallelism: A Survey

Jimcy Babu[1], Kavitha V.[2]
M Tech, CMRIT, Bangalore, Ph D Scholar, Jain University, Bangalore

Abstract: Recently proposed processor micro-architecture that generates high Memory Level Parallelism promise substantial performance gains. The performance of memory bound commercial applications such as databases is limited by increasing memory latencies. The ever increasing computational power of contemporary microprocessors reduces the execution time spent on arithmetic computations significantly. This paper reviews various techniques to leverage Memory Level Parallelism. Techniques reviewed in this paper are Code Transformations, Dynamic Warp Subdivision and Recovery free Value prediction. The survey reveals that, Code Transformation had an execution time reduction averaging of 20% in a multiprocessor and 30% in uniprocessor; Dynamic Warp Subdivision had improved performance on average by 15% on bulk synchronous cache organization with a maximum speed up of 1.6X and 17% on a coherent cache hierarchy with a maximum speed up of 1.9X and Recovery free Value prediction enhances Memory Level Parallelism effectively and achieves significant speed ups.

Keywords: Memory Level Parallelism (MLP); Code Transformation(CT); Out-of-order issue; Instruction Level Parallelism(ILP); Pre-fetching; warp-split table (WST).

I. Introduction

As the imbalance between processor and memory speeds increases, the focus on improving system performance moves to the memory system. Currently, processors are supported by large on-chip caches that try to provide faster access to recently-accessed data. Unfortunately, when there is a miss at the largest on-chip cache, instruction processing stalls after a few cycles, and the processing resources remain idle for hundreds of cycles. The inability to process instructions in parallel with long-latency cache misses results in substantial performance loss. One way to reduce this performance loss is to process the cache misses in parallel. Techniques such as non-blocking caches, out-of order execution with large instruction windows, run-ahead execution and pre-fetching improve performance by parallelizing long-latency memory operations. The notion of generating and servicing multiple outstanding cache misses in parallel is called Memory Level Parallelism (MLP). Memory Level Parallelism (MLP) is a term in computer architecture referring to the ability to have pending multiple memory operations, in particular cache misses or transition look-aside buffer misses, at the same time. In single processor, Memory Level Parallelism (MLP) may be considered a form of Instruction Level Parallelism (ILP). However, Instruction Level Parallelism is often mixed up with superscalar, the ability to execute more than one instruction at the same time. Example: a processor such as the Intel Pentium Pro is five ways superscalar, with the ability to start executing five different micro-instructions in a given cycle, but it can handle four different cache misses for up to 20 different load micro-instructions at any time. It is possible to have a machine that is not superscalar but which nevertheless has high Memory Level Parallelism. Arguably a machine that has no Instruction Level Parallelism, which is not superscalar, which executes one instruction at a time in a non-pipelined manner, but which performs hardware pre-fetching (not software instruction level pre-fetching) exhibits Memory Level Parallelism (due to multiple pre-fetches outstanding) but not Instruction Level Parallelism. This is because there are multiple operations outstanding, but not instructions. Instructions are often mixed up with operations.

II. Techniques to increase Memory Level Parallelism

A. Code Transformations (CT)

Current commodity microprocessors improve performance through aggressive techniques like multiple instruction issue, out-of order (dynamic) issue, non-blocking reads, and speculative execution to exploit high levels of instruction-level parallelism (ILP). Although ILP techniques successfully and consistently reduced the CPU component of execution time, their impact on the memory (read) stall component was lower and more application-dependent, making read stall time a larger bottleneck in ILP-based multiprocessors than in previous-generation systems. In particular, current and future read miss latencies are too long to overlap with other instruction types. Thus, an ILP processor needs

to overlap multiple read misses with each other to hide a significant portion of their latencies. An out-of-order processor can only overlap those reads held together within its instruction window. Independent read misses must therefore be clustered together within a single instruction window to effectively hide their latencies (read miss clustering or clustering). Most of the applications typically did not exhibit much read miss clustering, leading to poor parallelism in the memory system. Code transformation can increase parallelism in memory systems with out-of-order processors, by overlapping multiple read misses within the same instruction window, while preserving cache locality.

i) **Read miss clustering**

To understand the sources of poor read miss clustering in typical code, we consider a loop nest traversing a 2-D matrix. Figure 1 graphically represents three different matrix traversals. The matrix is shown in row major order, with crosses for data elements and shaded blocks for cache lines. Figure 2 relates these matrix traversals to code generation, with pseudo-code shown in row-major notation.

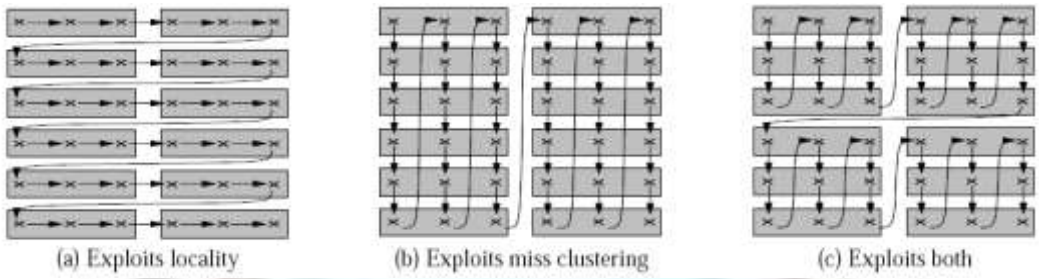

Figure 1: Impact of matrix traversal order on miss clustering. Crosses represent matrix elements (row-major order), and shaded blocks represent cache lines. (Ref 1)

```
for(···j++)
  for(···i++)
    ···A[j,i]
```
(a) Base code

```
for(···i++)
  for(···j++)
    ···A[j,i]
```
(b) Interchange

```
for(···jj+=N)
  for(···i++)
    for(j=jj;j<jj+N;j++)
      ···A[j,i]
```
(c) Strip-mine and interchange

```
for(···j+=N)
  for(···i++){

    ···A[j,i]
    ···A[j+1,i]
    ......
    ···A[j+N-1,i]
```
(d) Unroll-and-jam

Figure 2: Pseudo-code for Figure 1 matrix traversals (row-major notation). [Ref 1]

Unroll-and-jam is preferred instead of strip-mine and interchange for two reasons.

- First, unroll-and- jam allows us to exploit additional benefits from scalar replacement.
- Second, unroll-and-jam does not change the inner-loop iteration count.

The shorter inner loops of strip-mining can negatively impact techniques that target inner loops, such as dynamic branch prediction. By increasing inner-loop computation without changing the iteration count, unroll-and-jam can also help software pre-fetching [4].

ii) Dependences that limit memory level parallelism

Three kinds of limitations to read miss parallelism are identified as

- cache-line dependences,
- address dependences, and
- Window constraints.

iii) Performance Analysis

Clustering transformation is evaluated using a latency-detection micro-benchmark and five scientific applications. Table 1 summarizes the evaluation workload for the simulated and real systems. Lat-bench is based on the lat mem rd

kernel of lm-bench [5]. Lat-bench is clustered with unroll-and-jam. The base Lat-bench indicates an average read miss stall time of 171 ns on the simulated system. Clustering drops the average stall time caused by each read miss to 32 ns, a speedup of 5.34X. On the Convex Exemplar, clustering reduces the average stall time for each miss from 502 ns to 87 ns, providing a speedup of 5.77X.

	Simulated system		Convex Exemplar	
Microbenchmark	**Input Size**	**Procs.**	**Input Size**	**Procs.**
Latbench	6.4M data size	1	40M data size	1
Application	**Input Size**	**Procs.**	**Input Size**	**Procs.**
Em3d	32K nodes, deg. 20, 20% rem.	1,16	100K nodes, deg. 20, 20% rem.	1,8
Erlebacher	64x64x64 cube, block 8	1,16	128x128x128 cube, block 8	1,8
FFT	65536 points	1,16	4M points	1,8
LU	256x256 matrix, block 16	1,8	4224x4224 matrix, block 128	1,8
Mp3d	100K particles	1,8	2M particles	1
MST	1024 nodes	1	1024 nodes	1
Ocean	258x258 grid	1,8	1026x1026 grid	1,8

Table 1: Data set sizes and number of processors for experiments on simulated and real systems [Ref 1].

Figure 3 shows the impact of the clustering transformations on application execution time for the base simulated system. Figure 3(a) shows multiprocessor experiments, while Figure 3(b) shows uniprocessor experiments. The execution time of each application is shown both before and after clustering (Base/Clust), normalized to the given application and system size without clustering.

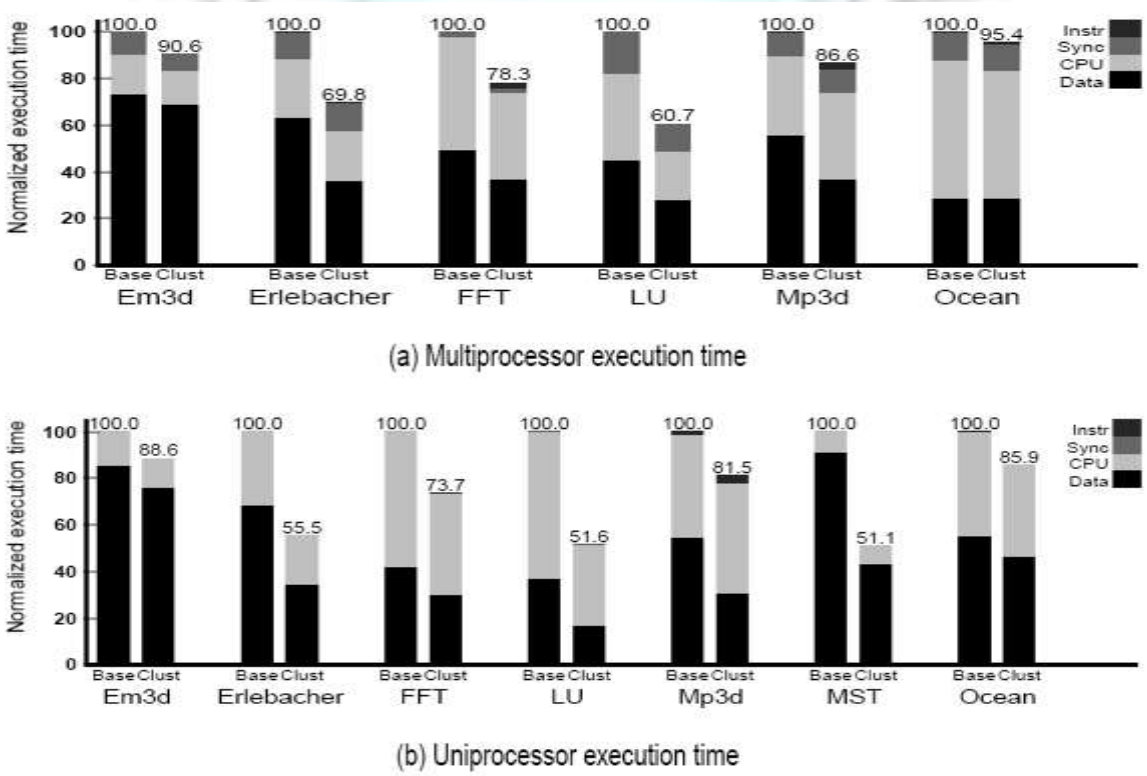

Figure 3: Impact of clustering transformations on application execution time.[Ref 1]

Overall, the clustering transformations studied provide from 5–39% reduction in multiprocessor execution time for these applications, averaging 20%.

% execution time reduced	Em3d	Erlebacher	FFT	LU	Mp3d	MST	Ocean
multiprocessor	9.2	21.4	16.6	22.7	N/A	N/A	-2.9
uniprocessor	13.0	34.3	28.9	23.8	21.7	38.1	21.6

Table 2 : Impact of clustering transformation on Convex Exemplar execution time.[Ref 1]

There are numerous differences between the simulated system and the real machine. Table 2 shows that clustering also provides significant benefits in the real system. The multiprocessor and uniprocessor execution time reductions from clustering range from 9–38% for all but the multiprocessor version of Ocean, which sees 3% degradation.

B. Dynamic Wrap Subdivision

To mitigate the penalty of long latency memory accesses with limited number of warps, dynamic warp subdivision allows threads that hit during divergent cache accesses to continue and exploit more MLP. This technique is named as MLP-aware warp subdivision (MAWS). The split warps after the subdivisions are referred to as warp-splits. Threads that continue after they hit the cache are named as run-ahead threads and they form a run-ahead warp-split. Threads that stall due to cache misses are called fall-behind threads and they form a fall-behind warp-split. The original warp is regarded as the root warp-split and it can be subdivided recursively.

i) Exploiting MLP

Figure 4 compares conventional SIMT execution with MAWS upon divergent cache-accesses. Consider a warp that has two memory access instructions which would both incur diverged cache misses in the conventional execution model. Assuming all other warps have been suspended already, MAWS can avoid stalling (Figure 4(b)(i)) or reduce the stalling cycles (Figure 4(b)(ii)) in two scenarios:

- With conventional SIMT execution, the fall-behind threads would hit the cache upon the latter instruction anyway, and it is the run-ahead threads that now miss the cache. In this case, MAWS allows run-ahead threads to issue their memory requests earlier.

- With conventional SIMT execution, the fall-behind threads would miss the cache upon the latter instruction, and they request the same cache block as some of the run-ahead threads. In this case, the run-ahead warp-split plays the role of pre-fetch threads for the fall-behind warp-slit. Different from speculative precomputation or run-ahead simultaneous threads [6] in the context of SMT, the run-ahead warp-split always perform useful computation and threads' states are saved right away, requiring no ROB or dependency analysis that would otherwise complicate the design of the simple, in-order Warp Processing Unit (WPU).

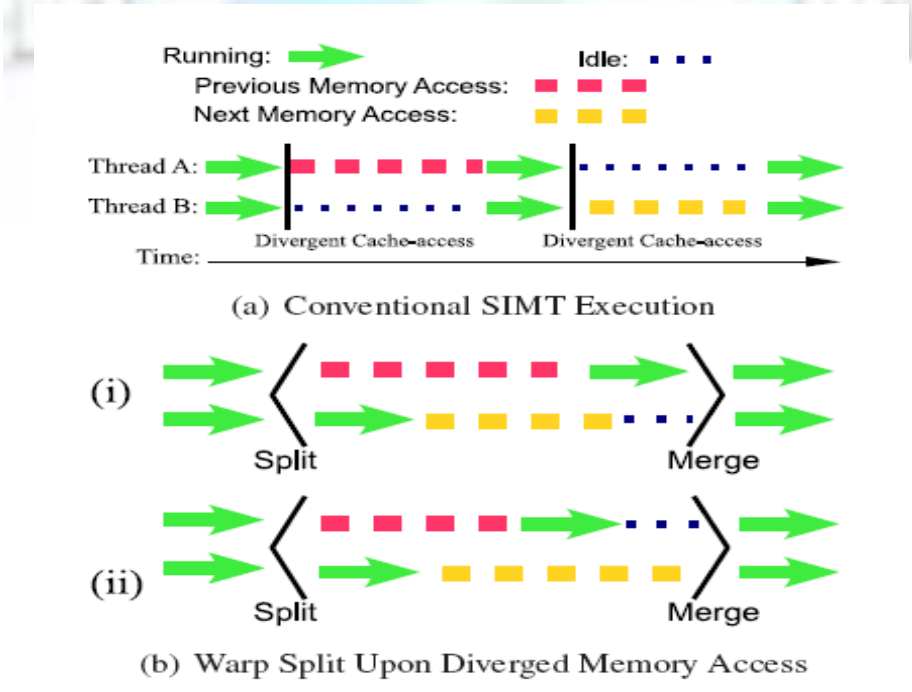

Figure 4: Comparing (a) conventional SIMT execution with (b) MAWS using a simplified WPU model with all its warps stalled due to cache misses except for one.[Ref 2]

In both cases, the long latency memory request which would otherwise stall the pipeline is issued earlier than they would in the conventional execution model. In consequence, the pipeline stalls for fewer cycles and performance can be improved.

ii) Implementing Warp-splits

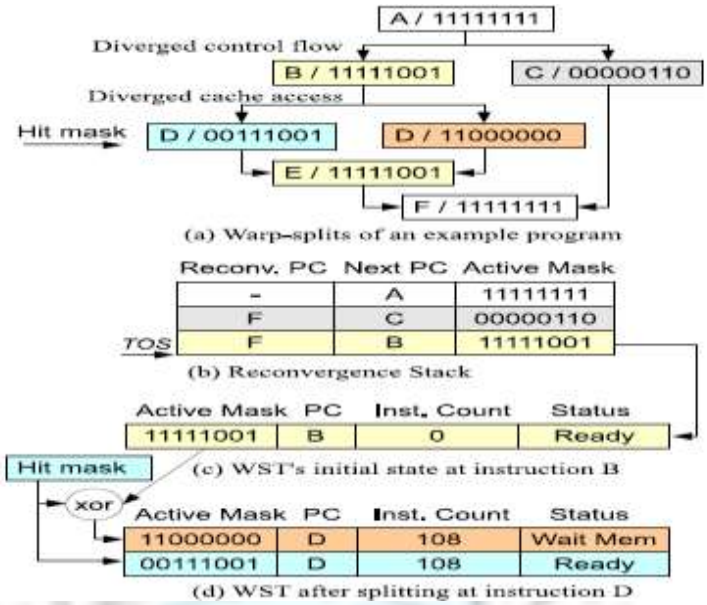

Figure 5: Warp splitting upon diverged cache accesses and its merging process. [Ref 2]

Hardware implementation uses a warp-split table (WST) for each warp to keep track of all its existing warp-splits. A WST entry records a warp-split's current PC, its instruction count, executing status, the priority, and the active mask with set bits denoting the belonging threads. Both the PC and the instruction count are used for the purpose of merging the warp-splits. The executing status and the priority are used for scheduling, and the active mask is used for selecting the threads in the corresponding warp-split to run in SIMT. The post-dominator based re-convergence scheme is used to handle conditional branches [7]. Figure 5: shows the Warp splitting upon diverged cache accesses and its merging process.

iii) Performance Improvement

Lat-Spec with Loop-Bypassing and Shallowest-Warp-First (SWF) scheduling works consistently well across all applications without any degradation. Comparison of speedups resulted from all combinations of the optimization techniques involved in MAWS is shown in figure 6.

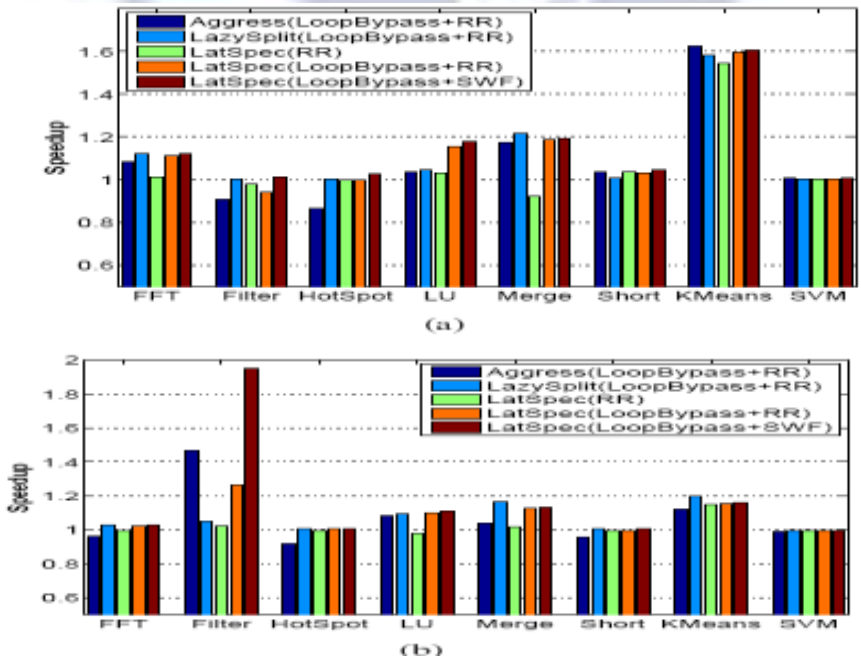

Figure 6: Speedup of various MLP optimizations on (a) a bulk synchronous, one-level cache organization; and (b) a two-level coherent cache hierarchy (Ref 2).

The combination includes subdivision strategies (Aggress, Lazy-Split, and Lat-Spec), loop bypassing, and scheduling policies (RR, SWF). The typical combinations are listed in Figure 6. For the bulk-synchronous cache organization, Lat Spec (Loop-Bypass + SWF) outperforms Lazy-Split (Loop-Bypass + RR) significantly with LU and it leads to an average performance improvement of 15%, compared to Lazy-Split's performance gains of 12%. On the two-level cache hierarchy, Lat-Spec (Loop-Bypass + SWF) and Lazy-Split (Loop-Bypass + RR) achieve a performance improvement of 17% and 7%, respectively; no performance degradation is observed for both systems. On the other hand, although Aggress is able to achieve speedups for several applications, it leads to performance degradation up to 13% in the bulk-synchronous cache organization and 8% in the two-level cache hierarchy, which is caused by pipeline under-utilization due to narrow warp-splits that are not merged in time.

C. Recovery Free Value Prediction

Recovery Free Prediction is a novel technique to parallelize sequential cache misses speculatively. The target workload is memory intensive workloads with heavy pointer chasing. The idea is developed upon value prediction, which was originally proposed as an instruction level parallelism (ILP) optimization to break true data dependencies in computations. Since the data dependence between pointer chasing loads enforces the sequential execution, value prediction has the capability to parallelize these loads, thereby increasing the memory level parallelism (MLP). So, for memory intensive applications the largest performance potential of value prediction lies in its capability to enhance MLP instead of ILP.

i) Value Prediction

Since the focus is on using value prediction to increase MLP, the hardware overhead to support value prediction and value speculative execution can be significantly reduced. In this, authors [3] propose to use value prediction only for pre-fetching so that the complex value prediction validation and misprediction recovery mechanisms are avoided and only minor changes in the hardware are necessary.
Values produced by individual instructions exhibit localities and different value prediction schemes are proposed to exploit such localities to break true data dependencies. In a typical value prediction/speculation scheme proposed for a superscalar processor, the prediction of an instruction enables its dependent instructions to be executed speculatively. If the prediction is wrong, however, a recovery scheme is necessary to squash the speculative results and to re-execute these affected instructions with correct data. For memory intensive workloads with heavy pointer chasing, sequential cache-misses resulting from pointer chasing code structures dominate the overall execution time. These cache misses form a memory dependence chain since one missing load's address is dependent on the previous missing load's value.

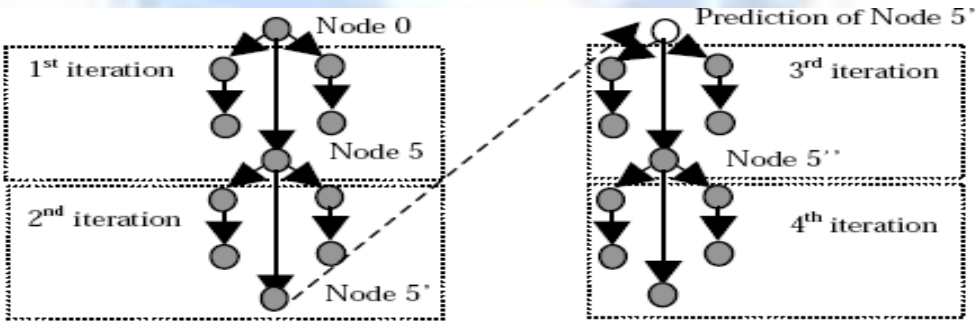

Figure 7: Predicting the value of Node 5' enables overlapping of cache misses in different iterations [Ref 3].

The example in Figure 7 illustrates that the effectiveness of value prediction in breaking the true memory dependence chain so that sequential cache misses can be processed in parallel and MLP can be enhanced. Such effectiveness is affected by several characteristics of the memory dependence chain.

- The first is the length of the memory dependence chain.
- The second is which missing load along the dependence chain is predicted.
- The third is the predictability of these missing loads' values since more accurate prediction will result in more useful speculative executions.

It is found that value prediction can be more effective than traditional address prediction based pre-fetching techniques for the same predictability model. The main reason is that while pre-fetching techniques only bring the data close to the processor (e.g., the L1 D-cache), value prediction takes one step further by using the fetched data to drive other dependent load instructions to be executed early.

ii) Recovery Free value Prediction

Value prediction has great potentials to enhance MLP by overlapping otherwise sequential cache misses. To implement such a technique, however, complex hardware support is necessary to validate the prediction and to perform recovery from value mis-predictions. To support recover-free value prediction, only minor hardware changes are necessary. The proposed design is based on a MIPS R10000 style micro-architecture [8], which has a 7-stage pipeline as shown in Figure8.

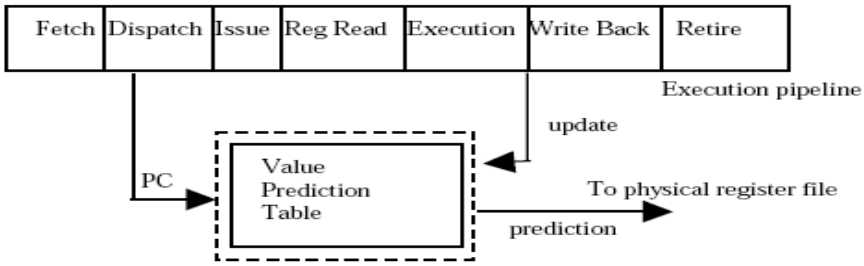

Figure 8: The execution pipeline. [Ref 3]

There are four key changes to the hardware, presented as follows:

- First, a value predictor is included in the front-end of the processor and is indexed with pc.
- Secondly, two flag bits are added to control value speculative execution. One flag bit, called value prediction speculative (vp), is added to every entry of issue window or RUU. The other flag bit, called value prediction ready (vp_ready) , is added for each register in the physical register file.
- Thirdly, the instruction selection logic is modified so that it prioritizes the issue of un-speculative instructions and prohibits the speculative execution of store and branch instructions.
- Fourthly, to break the alias (i.e., load-after-store) dependencies, the vp flag is set for the load instructions that are stalled due to prior unresolved store addresses.

iii) Performance Evaluation

With recover-free value prediction, the overall execution time is significantly reduced and MLP is much improved. Figure 9 shows the speedups of the proposed recovery-free value prediction and it shows that the proposed technique achieves significant speedups for memory intensive benchmarks, from 3.2% for the benchmark health to 24% for the benchmark mst. For the well-known pointer-chasing benchmark mcf, the speedup is 19.6%. For computation intensive benchmarks, smaller speedups (average of 0.5%) result, which is expected since the reduction in the D-cache miss rate for these benchmarks is small. The only benchmark that shows a negative speedup (-0.7%) is gcc.

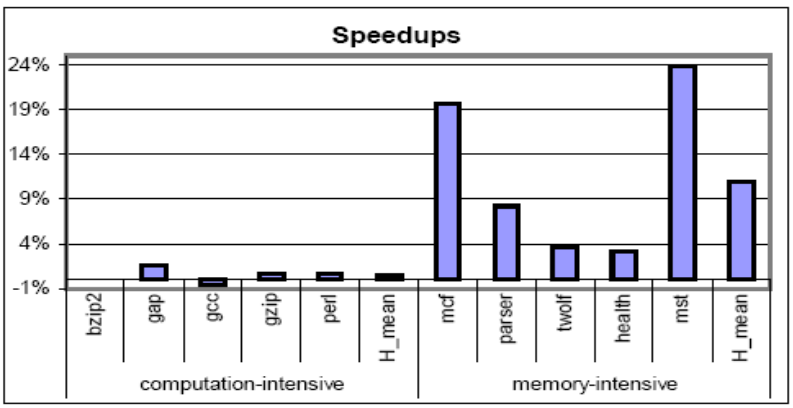

Figure 9: The speedups of using recovery-free value prediction [Ref 3].

Comparing proposed recovery-free scheme to the traditional value prediction, it the traditional value prediction achieves higher speedups for computation intensive benchmarks. For memory-intensive benchmarks, recovery-free prediction scheme has much higher speedups since it avoids the misprediction penalties and benefits from speculative memory disambiguation.

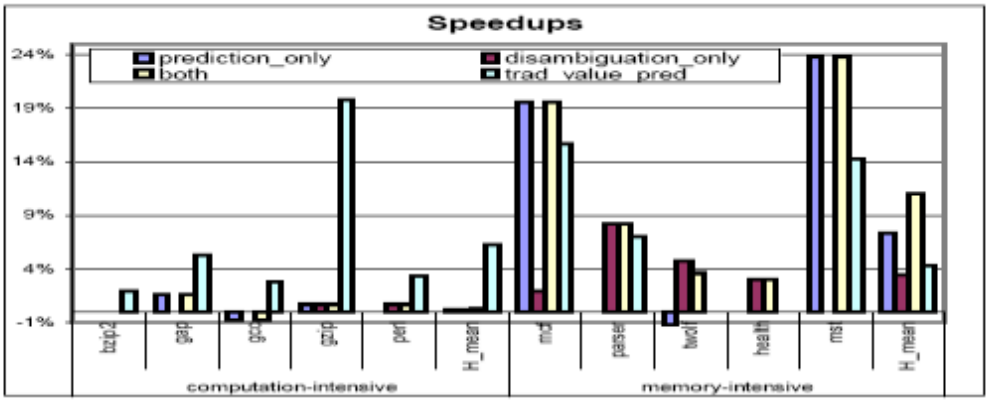

Figure 10: The speedups resulting from breaking different dependencies and traditional value speculation [Ref 3].

In recovery-free value prediction, the value predictor is updated with un-speculative execution results (i.e., the computation results not involving direct/indirect predicted values), thereby being able to achieve higher prediction accuracies than the traditional value speculation scheme. The results in Figure 10 also suggest another interesting optimization: we can apply recovery-free value prediction selectively by monitoring the dynamic behaviour of a workload. Only if the workload is memory intensive (e.g., the L1 D-cache miss rate is larger than 10%), the recovery-free value prediction is turned on. Otherwise, recovery-free value prediction is turned off or only the aggressive memory disambiguation is used for pre-fetching.

Acknowledgment

I would like to express my sincere thanks to my guide Mrs. Kavitha. V, Associate Professor, Electronics and communication, CMRIT Bangalore, for the timely guidance and encouragement to make this review, a success.

Conclusion

Three Techniques to leverage Memory Level Parallelism has been analyzed in this paper. Each one has its own advantages as well as disadvantages. Code Transformations can improve Memory Level Parallelism in systems with out-of order processors. Compiler transformation known for other purposes has been adapted to the new goal of memory parallelism. The experimental results show substantial improvements in memory parallelism, thus hiding more memory stall time and reducing execution time significantly.

MLP-Aware Warp Subdivision (MAWS) is used to mitigate the penalty of divergent cache-accesses, which leverages MLP by subdividing warps upon divergent cache-accesses and allow threads that hit the cache to run ahead and issue more memory requests. Lazy Split subdivides warps only when no more warps can proceed and exploit more MLP, and Latency-speculating Split reduces the number of unnecessary subdivisions when run-ahead warp-splits are not likely to be beneficial. Furthermore, loop bypassing improves the ability of run-ahead warp-splits to proceed across loop boundaries. On average, this technique improves the performance by 17% on the coherent cache hierarchy and 15% on the bulk-synchronous cache organization.

Value prediction can enhance MLP for memory intensive benchmarks with heavy pointer chasing. Value prediction is proposed for microprocessors with long memory latency operations for data pre-fetching so that complex prediction validation and misprediction recovery mechanisms are avoided and only minor hardware changes are needed. The same hardware changes enable aggressive memory disambiguation for pre-fetching. Recovery-Free Value prediction technique enhances Memory Level Parallelism effectively for a wide range of benchmarks and achieves significant speedups.

References

[1]. Vijay S.Pai and Sarita Adve, “Code Transformations to improve Memory Level Parallelism,”Copyright IEEE 1999.
[2]. Jiayuan Meng, David Tarjan and Kevin Skadron,”Leveraging Memory Level Parallelism using Dynamic Warp Subdivision,”University of Virginia Department of computer science Tech Report CS-2009-02,April 2009.
[3]. Huiyang Zhou and Thomas M. Conte, “Enhancing Memory Level Parallelism via Recovery Free Value Prediction,”Department of Electrical and computer Engineering, North Caroline State University,1-919-513-2014.
[4]. V. S. Pai and S. Adve, “Improving Software Prefetching with Transformations to IncreaseMemory Parallelism,”Tech. Rep. 9910, Department of Electrical and Computer Engineering, Rice University, November 1999.

[5]. L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in Proceedings of the 1996 USENIX Technical Conference, pp. 279–295, January 1996.
[6]. T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead threads to improve smt performance. HPCA '08, pages 149–158, Feb. 2008.
[7]. W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In MICRO '07, pages 407–420, Washington, DC, USA, 2007.
[8]. K. C. Yeager, "The MIPS R10000 superscalar microprocessor", IEEE Micro, 1996.
[9]. www.wikipedia.org