MATCH: Memory Address Trace CacHe

Rice University – ELEC 525 Final Report
Noah Deneau, Michael Haag, David Leal, and Arthur Nieuwoudt

***Abstract* — In this paper we explore methods for hiding memory latency. We analyzed the memory traces of several applications and determined that patterns exist with little spatial locality. Therefore, these patterns will perform poorly in the L1 data cache. With increasing memory latency, it is important to capture and store these patterns in order to feed data to wide-issue superscalar processors.**

To exploit these patterns, we propose a novel architecture, the Memory Address Trace CacHe (MATCH), that captures and stores recurring memory access patterns. The architecture is composed of three primary structures located off the critical path: the Pattern Generator, Pattern Buffer and MATCH Cache. Using SimpleScalar, we simulated 5 SPEC2000 benchmarks on our architecture to gauge performance relative to standard cache structures. Our results indicate that MATCH significantly improves performance on Ammp, which has speedups ranging from 24 to 43 percent. These results demonstrate that MATCH has potential but requires further refinement.

***Index Terms* — memory traces, trace cache, MATCH, prefetching**

I. Introduction

As companies continue to introduce higher speed processors into the market, the disparity between transistor switching times and memory access latency increases. To continue the performance improvement defined by Moore's Law, one must analyze and improve the system as a whole in order to obtain the desired result. To this end, steps must be taken to narrow the gap between memory latencies and processor frequencies. The relative increase in memory access times plays a key role in several major functions of the microprocessor.

In order to feed wide-issue the superscalar implementations prevalent toady, computer architects must deal with in memory latency in both the instruction and data realms. In short, a the processor must be able to fetch, decode, and issue enough instructions and access the appropriate data every cycle to utilize all of its available functional units.

In order to combat increasing instruction memory latency, researchers have focused on improving the cache structure of the microprocessor in an effort to hide some of the memory latency. Deeper and more complex cache structures have been implemented and proposed over the past few years. In addition, large issue buffers, more physical registers, and speculative fetch engines have all been examined in order to supply the execution core with the necessary, continuous-stream of instructions and the data on which the instructions operate.

After effectively creating a continuous supply of instructions to the core, one must next make sure that the instructions can access the necessary data in order to efficiently utilize the processor's functional units. We continue to see that, while speculation efforts have improved dramatically, the large data memory latency is increasingly the major bottleneck for modern computer systems. While the memory latencies at the supply side of the execution core have been decreased through a wide variety of new techniques, the latency to main memory during the execution of instructions has remained mainly dependent on brute-force improvements, such as improved DRAMs to provide faster buses and higher clocked chips or caches that are limited in size by increasing wire delay.

The remainder of the paper is organized as follows. In Section II we provide more insight into the motivation behind our memory address trace cache and discuss our hypothesis. In Section III we present the architecture for MATCH. Section IV focuses on our experimental methods. Information will be given on the simulations performed and the critical parameters examined. Section V will present our results, which include the performance and analysis of our simulations. We conclude in Section VI with an overview of our key results along with a discussion of the limitations in our

study and possible future work.

II. Motivation and Hypothesis

A. Motivation

To continue enhancing processor performance, the supply of instructions as well as data must scale with faster clocks and wide issue widths. Several studies have focused on increasing the number of instructions supplied to the processor by using alterative fetch and issue policies [1] or trace caches [2]. Data prefetching schemes related to our concept in hardware [4] and in the compiler [5] have also been investigated. Traditionally, these researchers approached the problem of the memory latency by implementing newer and more optimized cache hierarchies. However, these methods fail to address applications with poor spatial locality. In addition, as caches grow larger, longer blocks of contiguous data from memory are sent across the memory bus and occupy space in the cache, even when only a fraction of this data is required. These types of accesses waste cache space and memory bus capacity. In recent years, we have seen that the spatial and temporal locality of programs will no longer be able to hide this problem [3].

Coupled with the above difficulty is the fact that the programs running on today's machines are changing rapidly. The emergence of more parallelized applications, requires more noncontiguous accesses to memory. In these situations, reading in larger blocks from cache will waste precious system resources while providing only marginal benefits.

As systems adapt to these more parallelized operations, we will see larger number of functional units added to the execution core and more resources devoted to feeding these units with instructions. We feel that a more pressing issue is not the functional units, which are relatively cheap, nor the supplying of instructions, which has seen significant improvements over the past few years, but rather a fast and efficient method to feed data into the execution core from main memory.

In order to reduce some of the memory latencies, we propose utilizing the concept of a trace cache [2] but adapting it to provide traces of data memory access patterns. In order to be effective, however, we need to be able to adequately understand what, if any patterns, are available in the memory accesses of a wide range of applications. To further motivate the ideas for our project, we created and analyzed memory traces of five applications from the SPEC benchmark suite: Ammp, Vpr, Mcf, Equake, and Parser. Each application memory trace consisted of 10 samples of 100,000 memory accesses.

TABLE I
PATTERN STATISTICS FROM MEMORY TRACES

App	Ave Pattern Length		Ave Pattern Frequency		Ave % of Trace in a Pattern	
	BB	Jump	BB	Jump	BB	Jump
Ammp	3.90	12.73	30.03	23.13	2.84	16.58
Vpr	34.41	33.80	9.18	32.06	10.26	12.88
Mcf	3.16	12.84	2.38	2.30	0.37	0.73
Equake	8.58	115.34	94.84	315.11	10.13	5.98
Parser	3.00	104.72	2.00	28.02	0.0004	9.79

Table I summarizes the patterns found from our analysis of the memory traces. In order to adequately simulate the possible patterns that could be captured in hardware, we limited the pattern size in our analysis program to a maximum of 256 memory addresses and a minimum of three. Two different methods were used to classify a pattern. The first, backward branch, accumulates all memory addresses into a pattern until a backward branch occurs in the program counter. The second method, jump, accumulates addresses into a pattern until the difference between the previous and a current program counter exceeds a fixed value. Table I reveals that a large number of patterns exist in all of applications and, if they could be efficiently stored and examined, then they could efficiently cache common memory access patterns and help reduce memory latency.

B. Hypothesis

Non-contiguous memory accesses significantly increase average memory latency and result in the underutilization of the normal caches. We propose that a *Memory Address Trace Cache* (MATCH) will help solve this problem. MATCH stores the memory addresses of several non-contiguous memory accesses that a pattern generator identified as a pattern based on backward branch. Whenever the same initial memory address is accessed, the MATCH provides the information necessary to speculatively fetch data into a special cache. This allows the storage of many noncontiguous memory accesses, which waste system

resources in the normal cache, in a new memory address trace cache (MATCH).

This approach has several advantages. First it reduces overall memory latency by speculatively retrieving and storing data that is required by patterns that will most likely appear again. As long as the speculative memory accesses do not interfere with the programs' memory accesses, they have minimal cost since the pipelined memory system is idle. Finally, we feel that MATCH is well suited for applications that redundantly access large data sets.

By storing data access patterns with MATCH, the system will reduce the number of reads to noncontiguous memory. This will improve the overall performance of a wide range of applications and will not degrade the performance of general applications.

III. Architecture

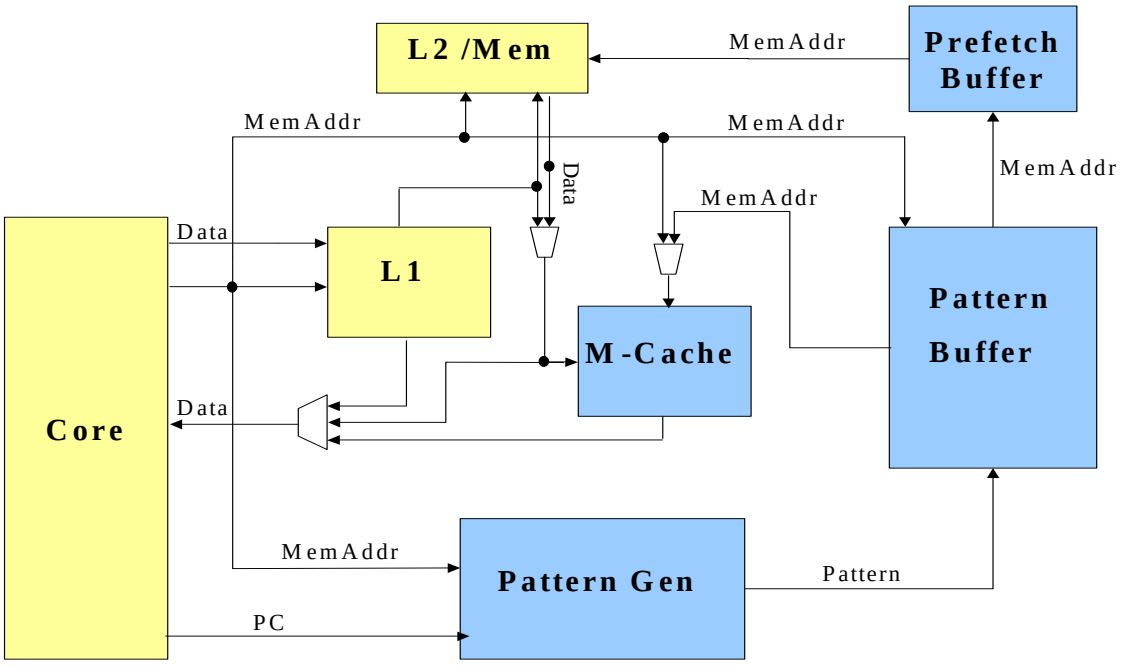
The MATCH architecture, depicted in Figure I, consists of three major components for detecting, storing and prefetching patterns. The Pattern Generator, described in section 2.1, looks at memory addresses and finds patterns that are sent to the Pattern Buffer, which holds recently used patterns and handles prefetching. Lastly, the MATCH Cache contains the data fetched via the Pattern Buffer, and is accessed on L1 data cache misses.

A. Pattern Generator

All memory loads are sent to the Pattern Generator and L1 data cache simultaneously as the operations issue from the load-store queue. The Pattern Generator stores each load's memory address into an internal buffer that builds continuously. "Hit/Miss" signals from the L1 cache are also routed to the Pattern Generator. Because the MATCH architecture is designed to primarily assist with patterns that the standard L1 cache cannot exploit, the Pattern Generator keeps a count of the L1 cache hits. It then compares this count with the length of the pattern found and makes a decision based on the L1 Cache Miss Threshold to forward the pattern to the Pattern Buffer unit or discard it. The L1 Cache Miss Threshold stipulates the percentage of L1 cache hits allowed in a valid pattern. For instance, if the Miss Threshold is .5, then the Pattern Generator only accepts patterns in which at most 50 percent of the memory accesses hit in the L1 cache while the Pattern Generator creates the pattern.

A pattern in the Pattern Generator's internal buffer is complete when either the buffer reaches its size limit, or when the Pattern Generator detects a backwards branch. We considered other methods of stopping a pattern, such as the program counter jumping past a certain threshold or finding a constant stride in the memory addresses. Based on our earlier memory trace simulations we determined that backwards branch was quite effective in finding loop branches and some procedure calls. It is also simple to implement, requiring only one comparison between the current and previous program

Figure I: MATCH Architecture Block Diagram



counters. Once the Pattern Generator has a complete pattern, it checks that the pattern is longer than the minimum allowed pattern length and that it passes the L1 Miss Threshold test described above. That is, a certain percentage of its memory accesses must have missed in the L1 cache. This prevents MATCH from storing patterns that can usually be retrieved from the L1 cache. A pattern that passes these tests is sent to the Pattern Buffer unit and the Pattern Generator is reset.

B. Pattern Buffer

The Pattern Buffer holds recently used patterns and handles data prefetching. It is constructed as an array of patterns received from the Pattern Generator, a row of state bits for the current pattern, and pointers to the current pattern and current address element in the pattern. Each pattern also has a "used" bit for a simple replacement policy.

A pattern is located in the Pattern Buffer simply by checking the first address. Thus, no two patterns can have the same starting address. From our analysis of memory traces, we determined that it would be faster to only check the first address, and surprisingly few different patterns detected by the Pattern Generator have the same starting addresses. When patterns arrive from the Pattern Generator, the Pattern Buffer searches the starting addresses of each pattern to determine if it already exists in the buffer. If a match is found whose used bit is not set, the row is replaced with the new pattern. If the starting address does not match any existing patterns, the pattern is inserted into an empty slot in the Pattern Generator, if one exists. If there are no empty spaces, the Pattern Generator evicts the first pattern it can find that has a "used" bit that is not set, i.e., the pattern has not yet been accessed. In the case that all patterns have been used at least once, all "used" bits are reset to zero and the pattern is inserted into the first location.

Each load issued from the load/store queue is also sent to the Pattern Buffer in parallel with the L1 cache and Pattern Generator. If the Pattern Buffer is not currently accessing any particular pattern, and a backwards branch has just occurred, the Pattern Buffer searches for the current load's memory address in its array. This is done by checking only the first addresses of each pattern, yet this process is quite slow since it must make one comparison for every row in the Pattern Buffer. Consequently, we only search for the start of a new pattern when a backwards branch occurs.

If a matching address is found in a row, the Pattern Buffer predicts that this load is the start of the pattern located in that row. It then begins to prefetch all addresses in the pattern into the MATCH Cache, and sets the used bit for that pattern. The addresses of subsequent memory accesses issued are checked to certify that they match in the current pattern. If an address does match the next address in the pattern, or if the final address has already been checked, the Pattern Buffer resets its current pattern and current address pointers and prepares to search for the next pattern.

Alongside the Pattern Buffer exists a buffer that handles the prefetching data from the memory addresses in patterns. As the Pattern Buffer works through a pattern, requests are sent to the Prefetch Buffer. The Prefetch Buffer, in turn, sends load requests to the L1 cache and simultaneously probes the MATCH Cache. Data does not need to be prefetched if it already exists in the L1 or MATCH cache. On each memory operation issued from the register update unit, the Prefetch Buffer checks to see if that memory address has arrived and been stored. If it has arrived, the address and data are stored from the Prefetch Buffer into the MATCH Cache.

C. MATCH Cache

The MATCH Cache, or M-Cache, is a hash-addressable set-associative cache that holds the data values that are prefetched by the Pattern Buffer. The organization is simple; it consists of an array of lists with elements containing the data values and a tag matching the corresponding memory address. The M-Cache can be thought of as a cache level between the L1 and L2 caches. Memory addresses are sent to it as soon as they are issued from the load/store queue, simultaneously with the request to the L1 cache. If the access misses in the L1 cache, the value from the M-Cache will be available immediately. In the case of an M-Cache miss, the data is retrieved from L2 memory. Meanwhile, the Pattern Buffer and Prefetch Buffer prefetch the pattern data values to fill the M-Cache. For store operations, the M-Cache follows a write-through policy for simplicity and speed. Since the L1 cache is always accessed first, this policy effectively handles any cache concurrency issues. The only exception occurs when a dirty value is evicted from the L1 cache, and the value also exists in the M-Cache. In this case, the value is written to the M-Cache as well as the upper levels of memory hierarchy.

TABLE II
SPEC2000 BENCHMARKS ANALYZED

Benchmark	Data Type	Description
188.ammp	Floating Point	Molecular dynamics simulation
181.mcf	Integer	Vehicle scheduling in public mass transportation system
197.parser	Integer	Syntactic parser of English language
175.vpr	Integer	Integrated circuit CAD design program
183.equake	Floating Point	Simulation of seismic wave propagation in large basins

IV. EXPERIMENTAL METHODOLOGY

In order to test our M-Cache structure described in Section III, as well as the Pattern Buffer and the Pattern Generator, we implemented them using the SimpleScalar 3.0 tool set because of its relevant basic architectural features and its support of SPEC benchmarks. After editing existing sections and adding the necessary functions to SimpleScalar, we compiled the code using GNU GCC 2.95.

SPEC2000 benchmarks were used to evaluate the performance of our architecture. These benchmarks consisted of 188.ammp, 181.mcf, 197.parser, 175.vpr, and 183.equake. We ran these benchmarks on UltraSparc 220R servers using the reduced data sets for a reasonable simulation time. These benchmarks provide a good mix of computations, including integer and floating point applications as well as scientific and non-scientific programs. A description of the SPEC2000 benchmarks we used in our analysis can be found in Table 2.

Before running the simulations on our M-Cache structure, we needed to establish a base case for comparison. We simulated the performance of the processor using only the baseline parameters found in Table 3 in the previous section. These simulations involved the basic SimpleScalar configurations with two caches, with the size of the L1 data cache set to 8KB. We also simulated a baseline comparison with the L1 data cache size set to 16KB to compare MATCH's performance with the typical solution: adding more cache capacity.

We compared the baseline architecture to our MATCH implementation. Our hardware structures were implemented with different configurations. We varied

TABLE III
BASELINE ARCHITECTURE CONFIGURATION IN SIMPLESCALAR

L1 Data Cache (using LRU eviction)	8K or 16K, 2-way set assoc
L1 Instruction Cache	16KB, direct-mapped
Unified L2 Cache	256KB, 4-way set assoc
L2 Cache Latency	9 cycles
Main Memory Latency	48 cycles
Memory Bus Width	8 Bytes
Issue Width	8
Branch mis-prediction latency	3 cycles
BTB	512 sets
Inorder	False
Register update unit size	32
Load Store queue size	16
Instruction TLB	16:4096:4:1
Data TLB	32:4096:4:1
Integer ALU's	5
FP ALU's	5
Memory Ports	3

the M-Cache size between 8KB and 16KB to investigate the optimal size of the M-Cache. The Pattern Buffer was also varied in size between 8KB and 64KB to determine whether an extremely large Pattern Buffer gives diminishing returns. We also wanted to observe whether patterns that reoccur frequently and are not evicted as often increase the performance of the MATCH architecture.

Pattern lengths were also fluctuated to demonstrate the effects of varying-sized patterns on the M-Cache performance. By varying this parameter we could analyze two different effects: (1) whether short patterns that occurred frequently are of any use in improving performance, and (2) whether larger patterns that were not found in the cache improved performance even if they were not found as frequently. We also experimented with different values for the L1 Cache Miss Threshold to determine the impact of accepting an assorted percentage of hits in the L1 cache. A summary of the MATCH simulation parameters can be found in Table 4. By varying these parameters of our hardware structures and analyzing the resulting simulations, we can develop the optimal configuration for the various applications.

While the SimpleScalar implementation is a fair model of a microprocessor architecture, several assumptions were made that should be taken into account. For example, the M-Cache access latency was assumed to be one cycle regardless of the varied sizes of the MATCH technology. This lack of additional L1

TABLE IV
MATCH SIMULATION PARAMETERS

Configuration	L1 Data Cache Size	Match Cache Size	Pattern Buffer Size	Maximum Pattern Length	Minimum Pattern Length	L1 Cache Miss Threshold
Baseline	8 KB			No MATCH Hardware Present		
BB	16 KB			No MATCH Hardware Present		
A	8 KB	8 KB	8 KB	32	6 – Ammp, 3 – All Other Benchmarks	.5 – Ammp, 1 – All Other Benchmarks
B	8 KB	8 KB	64 KB			
C	8 KB	16 KB	8 KB			
D	8 KB	16 KB	64 KB			

latency is intended to focus on the performance of the MATCH architecture and not on scalability issues. However, our experiments indicate that increasing the M-Cache latency to two cycles had a negligible impact on overall performance. Furthermore, in our prefetch implementations, we did not account for memory bandwidth limitations and instead assume infinite bandwidth. Also, no special penalties were examined or issued for the situation where a large amount of contention existed for the memory or system buses.

Although the Pattern Buffer can be large, the latency of the selection logic, which consists mostly of comparators, was not scaled to the Pattern Buffer size. Once again, this project focuses on the possible improvements in performance by the MATCH technology without spending a great deal of time to consider the impact of scalability.

The resulting IPC from each of the simulations was used as the definitive measure of performance when comparing the performance of the SPEC2000 benchmark simulations. We also used the calculation of IPC per cache miss rate to determine cache trends in our architecture.

From these simulations, one of the main goals is to reveal that the addition of our M-Cache structure will help to improve a system's performance by reducing the impact of memory latency. Using our baseline configurations as a basis for comparison, we will show in the following section that our M-Cache structure has positive results and the potential to improve on future microprocessor performance.

V. EXPERIMENTAL ANALYSIS

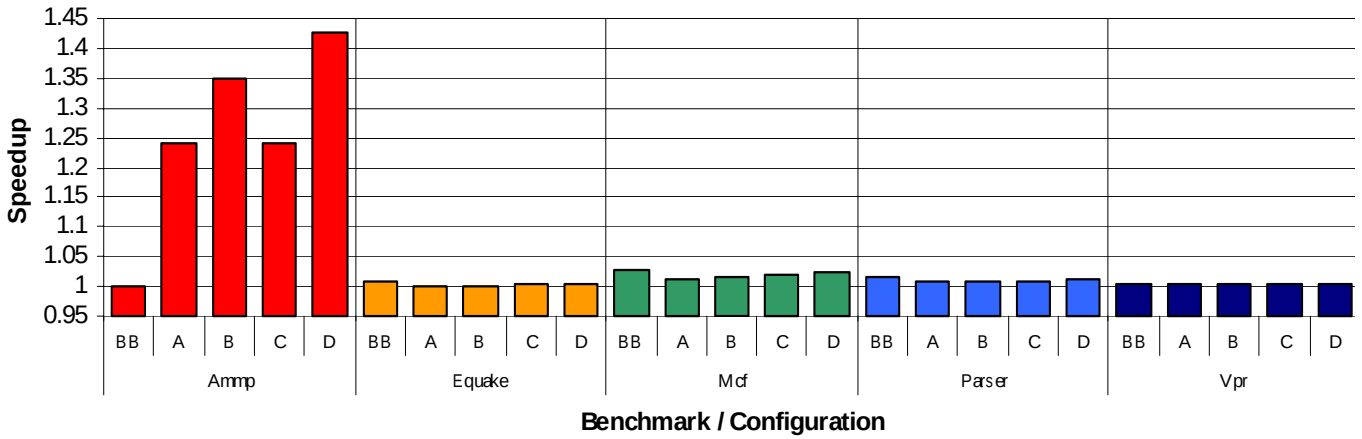
A. Results

Figure 2 shows the speedups for the 5 configurations in Table 4 relative to the baseline with an 8 KB L1 data cache. In all cases, MATCH outperforms the baseline configuration. MATCH improves the performance of Ammp by 24 to 43 percent over the baseline configuration, while simply adding more L1 cache provides little performance gain. Ammp solves large systems of ODEs, which require a significant number of regular, non-sequential array accesses. Therefore, Ammp exhibits regular memory access patterns, but these access patterns do not exhibit the spatial locality that a standard L1 cache can exploit. For this reason, the 16 KB data cache configuration (BB) provides only a .04 percent performance improvement. In contrast, the MATCH architecture was able to locate and store the appropriate patterns, which occurred frequently enough to provide significant performance gains.

The other benchmarks exhibit significantly less performance improvement as a result of the MATCH architecture. Mcf shows a 1 to 2 percent speedup, while the other benchmarks have speedups of less than 1 percent over the baseline configuration. With the exception of Ammp, the 16 KB L1 cache marginally outperformed all tested MATCH configurations.

The measured speedups also depend on the baseline L1 data cache miss rates, which are documented in Table 5. Since MATCH only improves memory access latencies of accesses that miss in the L1 data cache, the L1 cache miss rate has a significant impact on the absolute speedup numbers. The MATCH speedup to L1 data cache miss rate ratio tabulated in Table 5 indicates the normalized performance of MATCH for the five tested benchmarks. The normalized performance illustrates MATCH's performance in relation to the number of available long latency memory accesses for a given benchmark, which estimates how well MATCH is exploiting memory access patterns.

FIGURE II: MATCH PERFORMANCE



Like the absolute performance numbers, the normalized performance data indicates that Ammp achieves the greatest normalized performance. However, Ammp's normalized performance is only 6 times greater than the next highest benchmark, versus 24 times greater in terms of absolute performance. In contrast to the absolute performance numbers, Parser and Vpr surpass Mcf in normalized performance. Both Parser and Vpr have significantly lower L1 data cache miss rates than Mcf. Therefore, in both Parser and Vpr, MATCH is more effectively exploiting the applicable memory accesses than in Mcf. Equake exhibits the lowest normalized performance of the five benchmarks. Equake demonstrates virtually no performance improvement due to a large distance between repeated memory access patterns that are not captured in the L1 data cache. In fact, for Equake, the results indicate that out of approximately 7.5 million memory accesses the M-Cache only registered 400,000 hits.

The different MATCH configurations for Ammp have significant performance variation. Figure 2 also shows that increasing the Pattern Buffer size from 8 KB to 64 KB provides an additional 12 to 19 percent performance gain over the baseline configuration. This is due to the fact that the set of patterns generated is larger than the size of the Pattern Buffer. Since the Pattern Generator creates many patterns that are not utilized, the Pattern Buffer must be large enough to allow reoccurring patterns not to be replaced before they are executed. This effect seems to hold for the other benchmarks as well. Increasing the M-Cache size only improves performance when the pattern buffer contains enough patterns to exploit the increased M-Cache size. For instance, increasing the M-Cache size from 8 KB to 16 KB while keeping the Pattern Buffer size constant at 8 KB provides negligible performance improvement for Ammp. In contrast, increasing the M-Cache size by the same amount when the Pattern Buffer is 64 KB improves performance by 8 percent over the baseline configuration for Ammp. Unlike Ammp, Mcf performance improves when the M-Cache size is increased while keeping the Pattern Buffer size constant. Therefore, the degree to which the benchmarks exploit M-Caches of various sizes depends largely upon the ability of the Pattern Buffer to hold and prefetch valid, reoccurring memory access patterns, which varies by application.

The Pattern Buffer's ability to capture the desired reoccurring memory access patterns is directly related the Pattern Generator's pattern generation method and parameters. Figure 3 depicts the performance of Ammp and Mcf with several different L1 Cache Miss Thresholds. Ammp exhibits the greatest performance improvements when Miss Threshold is between .5 and .66. Mcf, in contrast, experiences the greatest performance when the Miss Threshold is 1. Ammp tolerates a lower Miss Threshold than Mcf because its higher L1 cache miss rate allows the Pattern Generator to generate a greater number of Patterns that miss in the L1 cache. Therefore, for optimal performance, the Miss Threshold must be set to allow the optimal number of patterns to be generated for a given application. Setting the Miss Threshold too low starves the M-Cache, while setting the Miss Threshold too high increases the number of unused patterns, which pollutes the Pattern Buffer. An unused pattern is a pattern that the application never repeats after generation and is eventually evicted from the Pattern Buffer. Therefore, the optimal L1 Cache Miss Threshold depends on the L1 data cache miss rate and the memory access patterns of a given application.

The Maximum Pattern Length also has a significant

FIGURE III: L1 MISS THRESHOLD USED IN PATTERN GENERATION

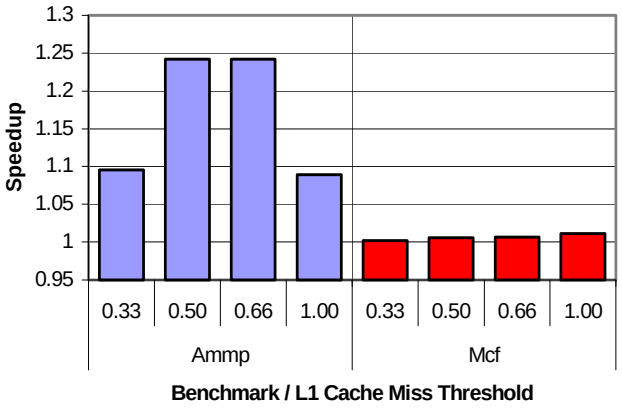
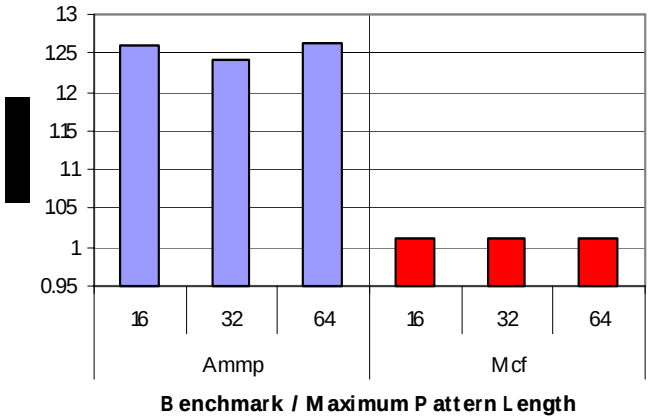


FIGURE IV: MAXIMUM PATTERN LENGTH



performance impact. Figure 4 shows the performance statistics for Ammp and Mcf for several various maximum pattern lengths. For Ammp, pattern lengths of 16 and 64 perform better than maximum pattern lengths of 32, while all maximum pattern length simulations perform almost equally well for Mcf. Therefore, like the L1 Cache Miss Threshold's behavior, the optimal Maximum Pattern Length depends on the application.

B. Hypothesis Evaluation

The experimental results support a qualified version of the original hypothesis. The patterns that the MATCH architecture exploits are present in every benchmark tested to some degree. In benchmarks with low L1 data cache miss rates, MATCH has few memory accesses with which to work, and consequently, the speedups are low. However, for benchmarks with higher L1 data cache miss rates, the speedups can potentially be impressive, as the results for Ammp indicate. Therefore, the hypothesis seems valid in programs that exhibit poor spatial locality and perform repeated work on fixed data sets.

C. Future Experimentation

The disparity between Ammp's performance and the performance of the other tested benchmarks fuels the need for further experimentation in order to fully evaluate the concept. The full memory trace and source code for Ammp needs to be analyzed to quantify the behavior that causes Ammp's large speedup. This information could then be used to identify other applications that could significantly benefit from MATCH. Furthermore, the simulation environment needs to be modified to reflect the unrealistic assumptions that were made in the simulations described in the Experiment Methodology section in order to validate Ammp's performance improvement.

Several modifications to the architecture should be made in order to account for a variable L1 Cache Miss Threshold and the Maximum Pattern Length. The Pattern Generator should dynamically record the number of patterns that it is generating and scale the L1 Cache Miss Threshold to allow only a certain number of patterns to be generated during a given interval of time. This would optimize the Pattern Buffer's utilization based on the currently running application. To more efficiently take advantage of the optimal maximum pattern length, the pattern buffer could be modified to have different regions that allow different length patterns. With this modification, the pattern buffer would waste less space for smaller patterns, thereby having more entries for a fixed size structure, which the results indicate would improve performance.

The set of applications tested in this paper provide insight into MATCH's performance on a wide range of application. This concept demands evaluation on a multitude of applications in order to identify the application types that benefit the most from the proposed architecture. Therefore, the applications tested were appropriate at this stage of the concept's analysis. From the five benchmarks it is hard to determine what class of applications benefit the most from MATCH. Equake and Ammp, the two floating-point benchmarks, have the best and worst normalized performance, respectively, while the integer benchmarks fall somewhere in the middle. The common characteristic applicable to all of the tested benchmarks is the relationship between absolute speedup and the L1 cache

miss rate. Therefore, additional benchmarks need to be analyzed in order to determine if other applications that have a high L1 cache miss rate benefit greatly from MATCH. Scientific applications may be a good starting point, since they tend to utilize the cache less effectively than their general-purpose counterparts. Therefore, further experimentation is necessary to conclude whether Ammp's impressive performance gains are common to other applications.

D. Cost Benefit Analysis

In order to conclusively determine if the cost of MATCH is worth its potential benefits, additional experimentation is required. If a certain class of applications has performance numbers similar to Ammp's, then the cost of MATCH is definitely worth the benefit for machine that run applications of that class. MATCH improved Ammp's performance by 24 percent in the relatively inexpensive configuration A and by 43 percent in the expensive configuration D, while simply doubling the L1 cache size provided little performance improvement. Therefore, for applications like Ammp, MATCH is a small price to pay for its impressive performance gains. However, if Ammp's speedups are atypical, MATCH is not worth the cost because the results indicate that a cheaper alterative, adding 8 KB additional L1 cache, improves performance to a greater degree. Therefore, the cost to benefit relationship for MATCH largely depends on the results of additional experimentation. However, if as few as 10 to 20 percent of typical applications can exploit MATCH as effectively as Ammp, MATCH is a win.

VI. Conclusion

The results indicate that MATCH has the potential to significantly improve performance on certain applications. As discussed in the Hypothesis Evaluation section, the results point to a revised version of the original hypothesis. While all applications should experience some performance improvement, only those applications with a large L1 miss rate will be significantly faster. Furthermore, the application must perform work on patterns that repeat often enough to be captured in a reasonable sized Pattern Buffer in order to experience a non-trivial speedup. Enough applications should exhibit these characteristics to make MATCH an effective addition to the cache hierarchy.

Additional experimentation needs to be performed to validate the revised hypothesis and improve upon MATCH. As discussed in the Future Experimentation section, the different methods of generating patterns and dynamically scaling pattern generation parameters need to be explored. In addition, more efficient Pattern Buffers should be explored to more effectively capture patterns of varying lengths. Furthermore, Ammp needs to be analyzed further to concretely establish what qualities make its performance far superior to the other benchmarks tested. Additional applications also need to be examined to determine if Ammp's performance is typical of other applications.

Because of the increasing impact of memory latency on processor performance, more complex methods of hiding latency are useful. MATCH, while adding significant hardware, has the potential to reduce the impact of memory latency, especially for programs that perform poorly in standard data caches. While research on exploiting data memory access patterns with poor spatial locality is still in its preliminary stages, MATCH's performance as well as the performance of other schemes [3, 4, 5] provide the incentive for further research. Additional experimentation and enhancement of MATCH will undoubtedly lead to useful methods for combating the memory latency that will plague future processors.

TABLE V: L1 DATA CACHE MISS RATES

Application	L1 Cache Miss Rate	MATCH Speedup / Miss Rate
Ammp	0.236	1.815
Equake	0.033	0.077
Mcf	0.136	0.185
Parser	0.034	0.302
Vpr	0.012	0.232

References

[1] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism," *IEEE*, pp. 24-33, 1997.

[2] Eric Rotenberg, Steve Bennett, and James E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 24-34, Dec. 1996.

[3] Impulse: An Adaptable Memory System, http://www.cs.utah.edu/impulse/

[4] Jean-Loup Bair and Tien-Fu Chen, " An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty", *Proceedings of Supercomputing*, November 1991.

[5] Todd C. Mowry, Monica S. Lam, Anoop Gupta, “Design and Evaluation of Compiler Algorithm for Prefetching”, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.