Improving Trace Cache Hit Rates using the Sliding Window Fill Mechanism and Fill Select Table

Muhammad Shaaban
meseec@rit.edu

Edward Mulrane
ted.mulrane@alum.rit.edu

Department of Computer Engineering
Rochester Institute of Technology
83 Lomb Memorial Drive
Rochester, NY 14623-5603

ABSTRACT

As superscalar processors become increasingly wide, it is inevitable that the large set of instructions to be fetched every cycle will span multiple noncontiguous basic blocks. The mechanism to fetch, align, and pass this set of instructions down the pipeline must do so as efficiently as possible. The concept of trace cache has emerged as the most promising technique to meet this high-bandwidth, low-latency fetch requirement. A new fill unit scheme, the Sliding Window Fill Mechanism, is proposed as a method to efficiently populate the trace cache. This method exploits trace continuity and identifies probable start regions to improve trace cache hit rate. Simulation yields a 7% average hit rate increase over the Rotenberg fill mechanism. When combined with branch promotion, trace cache hit rates experienced a 19% average increase along with a 17% average improvement in fetch bandwidth.

Categories and Subject Description

B.3.2 [**Memory Structures**]: Design Styles – *Cache Memory*
C.0 [**Computer System Organization**]: General – *System Architectures*

General Terms

Algorithms, Performance

Keywords

Superscalar processors, fetch mechanisms, trace cache, cache performance, fill mechanisms, branch promotion.

1. INTRODUCTION

One of the most substantial microarchitectural improvements since the early 1990s has been the introduction of multiple instruction issue in pipelined processors. As superscalar widths increase beyond the four and eight-way implementations of today, a proportional increase in execution bandwidth must be sustained by the rate of instructions fetched per cycle. This implies that the instruction fetch bandwidth must be at least equal to the number of instructions issued per cycle. Traditional superscalar fetch mechanisms can provide a set of instructions up to and including a single branch instruction. Furthermore, it has been stated that the average basic block size is five instructions for general-purpose code. Considering these two assertions, it is evident that fetch bandwidth has the potential to be a major bottleneck within the pipeline. Several mechanisms have been proposed to fetch past basic block boundaries, including the branch address cache and collapsing buffer. The most promising though has been the trace cache.

Storing instructions in dynamic execution order is the fundamental concept behind the trace cache. This contrasts with the traditional instruction cache, which stores instructions in static program order. The traditional trace cache relies on a multiple branch predictor to deliver up to *m* predictions per cycle. Several multiple branch prediction schemes have been proposed, often extending an established single branch prediction method. The trace cache is accessed in parallel with the I-Cache during the fetch stage, and uses the PC and the predictions to determined a hit or miss. If there is a hit, the corresponding trace is passed down the pipeline in its entirety. Otherwise, the core fetch unit retrieves instruction via the I-Cache as usual

The fill unit of the trace cache is responsible for populating the trace cache. Instructions are buffered as they are retired from the reorder buffer (or similar mechanism). When certain trace terminating conditions have been met, the contents of the buffer are used to form a new segment which is added to the trace cache. Two factors determine the effectiveness of the fill unit at generating relevant traces: How well *trace continuity* is maintained, and how well *probable entry points* are identified. The Sliding Window Fill Mechanim (SWFM) is a proposed method that exploits both these factors. The result is an increase in trace cache hit rates, which in turn improves fetch bandwidth. When this approach is coupled with branch promotion the hit and fetch rate improvement over the traditional trace cache is substantial.

The next section of this paper presents background information concerning the Rotenberg trace cache, branch promotion, the alternate fill mechanism, and other previously proposed fill unit enhancements. Section III presents the notions of trace continuity and probable entry points in further detail. The Sliding Window Fill Mechanism is described along with the Fill Select Table – an integral part of the scheme. A brief description of the benchmarks used, and the simulation results are presented in Section IV.

MSP'04', June 8, 2004, Washington, DC, USA.

2. BACKGROUND & PRIOR RELATED WORK

2.1 Rotenberg Trace Cache

Rotenberg et al [2] established the idea of trace cache in a comparison against other proposed mechanisms that aimed to solve the basic block fetch limitation. This work became the basis for a series of papers exploring various trace cache organizations and alternate methods. As presented in their paper, trace cache supplements the usual instruction cache by storing individual traces of dynamic instruction streams. Each line of the trace cache contains up to *n* instructions, and at most *m* basic blocks. The maximum number of basic blocks within a trace cache line is constrained by the throughput of the multiple branch predictor. The reason behind this constraint is the number of conditional branches in a trace cache line cannot exceed the number of parallel results from the predictor to which comparisons are made. Over the course of program execution, instructions that are retired are added to the fill buffer of the trace cache. When the number of instructions or basic blocks in the fill buffer meet *n* or *m* respectively, the group of instructions is added as a new line to the trace cache. An example trace cache line is shown in Figure 1. In addition to the instruction stream, additional information is recorded. Branch Flags indicate the path followed by each conditional branch within the trace. Since a variable number of branches could occur within a trace, a Branch Mask field must also be provided, indicating the number of branches (with a maximum of *m* - 1) and whether the trace ends in a branch. Two addresses are also stored, pointing to the next execution point following the last instruction of the trace. One is the Fall-through Address (corresponding to a not-taken result for a trace-terminating branch) and the other the Trace Target Address (corresponding to a taken result.) If the last instruction in the trace is not a conditional branch, both address fields point to the cache line starting with the next sequential instruction. In addition to the constraints *n* and *m*, an indirect jump or subroutine return also terminates a trace.

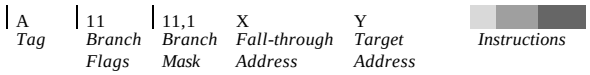

Figure 1: Example Trace Cache Segment

During instruction fetch, the trace cache is accessed in parallel with the I-Cache. When program execution presumably returns to an instruction that starts a cache line, the branches within the trace cache line are referenced with the results of the multiple branch predictor. If the predicted branch directions match each of the entries in the Branch Flags field, then the entire trace cache line can be fetched. If the predictions do not match the trace, then blocks are fetched from the instruction cache, and the trace cache line will be updated with the new stream of dynamic instructions.

2.2 Branch Promotion

The technique of branch promotion was proposed by Patel et al in [3]. Branch promotion exploits the fact that over half of conditional branches tend to be strongly biased during execution. By identifying these branches in hardware, they may be treated as statically predicted. These "promoted" conditional branches are marked via a single bit flag, and are associated with a taken or not-taken path. When the fill unit writes a cache line, the promoted branches are not included in the normal branch mask and flags fields. Thus, the predictor is alleviated from the overhead of storing the branch history for a promoted branch. This decreases aliasing within the predictor. If a static prediction proves incorrect (such as the final iteration of a loop) the machine is backed up to the end of the previous block and restarts along the correct path. The structure that decides to promote a branch to static prediction is the bias table. This table is a simple saturating counter indexed by the branch address (with tag compare.) The counter is incremented when the result of the branch is the same as the previous outcome. The fill unit will check the bias table against conditional branches in the retired instruction stream. If the count is greater than threshold value, the branch is promoted in the fill buffer. If the outcome of a promoted branch contradicts the static prediction twice in a row, the branch is demoted, and the branch bias table entry is cleared. Demotion upon two consecutive mispredictions was established to avoid the final iteration of a loop from demoting an otherwise strongly biased branch. Likewise, if the branch bias count falls below the threshold (the result of an alternating series of taken/non-taken results) the branch is demoted.

2.3 An Alternate Fill Mechanism

Prior to the initial Rotenberg et al paper on trace cache, a US patent was filed for a *Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line* [4]. Described is a mechanism that closely approximates the concept of the trace cache. One interesting aspect of this pioneering design is the fill unit policy. The Rotenberg scheme entails flushing the fill buffer to the trace cache once the maximum number of instructions (*n*) or basic blocks (*m*) has been reached. The next instruction to retire will be added to the empty fill buffer as the first instruction of a future trace. The fill method described in [4] differs by committing a trace line when *n* or *m* has been reached, then discarding the frontmost (oldest) basic block from the fill buffer and shifting the remaining instructions to free room for newly retired instructions. If effect, every new basic block encountered in the dynamic instruction stream causes a new trace to be added to the cache.

2.4 Other Proposed Fill Unit Enhancements

A few other schemes have been proposed that are related to the Sliding Window Fill Mechanism. Ramirez et al [5] propose a scheme that targets redundancy between the I-Cahce and trace cache. Called selective trace storage, it takes advantage of a core fetch unit that can fetch beyond not-taken conditional branches. This allows segments that consist of only sequential instructions or not-taken branches to be excluded from the trace cache without hindering fetch bandwidth. Consequently, trace cache utilization is reduced which reduces destructive aliasing and improves hit rates.

Trace preconstruction [6] is a rather elaborate method that is analogous to instruction pre-fetching in the traditional memory hierarchy. The goal of the preconstruction algorithm is to build probable traces prior to the point where they are utilized. Determining these traces is achieved by identifying regions of code that will be encountered later in the course of normal execution. The work by Jacobson and Smith relates to this proposal in that the utilization of trace start points in the fill unit is similar to the notion of region start points used during precontruction.

Lastly, an entire set of techniques known as instruction pre-processing optimizations can be applied to trace cache segments

as part of the fill unit logic. A few of these possible optimizations are discussed in [7], and demonstrate another way the fill unit can be used to improve performance.

3. THE SLIDING WINDOW FILL MECHANISM

3.1 Trace Continuity and Probable Entry Points

This sub-section serves as a precursor to the next by explaining two attributes of traces segments and trace cache: *Trace Continuity* and *Probable Entry Points*. The issue of trace continuity is best explained by way of example. Consider a set of basic blocks (A, B, C, D and E) that constitute a dynamic path free of indirect jumps. Each block is sized such that two blocks can occupy the fill buffer without exceeding either the *n* or *m*-constraint. Three blocks will not fit in the buffer though, as the number of instructions occupying the buffer will reach *n* before the entire third block can be added. This will result in a finalized trace that ends with a sequential instruction (opposed to a control instruction). If such a trace begins with block A, it will be followed by block B and the start of block C (identified as "C1"). Similarly, if the remainder of block C appears independent of C1, it will be identified as "C2". Figure 2 illustrates the fill process for both the Rotenberg and Alternate fill schemes.

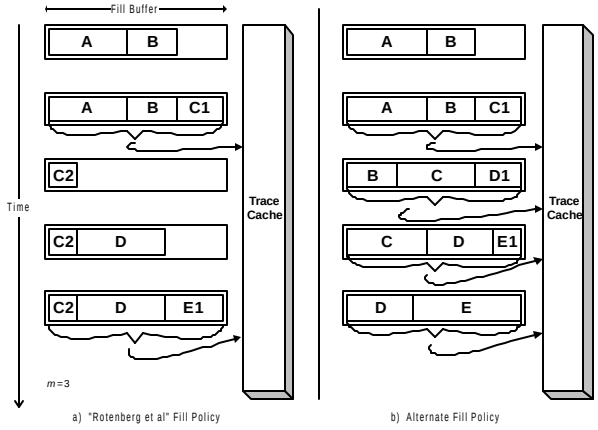

Figure 2: Fill Unit Example

Table 1 lists the resulting traces generated for each scheme above.

Table 1: Resulting Trace Segments

Rotenberg Fill Policy	Alternate Fill Policy
A-B-C1	A-B-C1
C2-D-E1	B-C-D1
	C-D-E1
	D-E

The issue of trace continuity becomes evident when the PC returns to the start of Block A. In either scheme, the trace A-B-C1 is fetched from the trace cache and passed down the pipeline. The PC will then be set to the address of C2. The Rotenberg scheme will generate another hit (C2-D-E1). The trace cache implementing the alternate fill scheme will experience a miss, as it does not store traces beginning with instructions that dynamically follow sequential instructions. The Rotenberg method therefore provides better trace continuity. It should also be clear that trace continuity is applicable only when dealing with *n*-constrained traces. If a trace is terminated by way of the *m*-constraint or indirect jump, the Trace ID of the subsequent trace will start a basic block, which does not pose a problem for either fill approach. Programs consisting of traces that are largely *n*-constrained are subject to decreased performance using the alternate fill scheme. This is a result of poor trace continuity.

Despite its deficiency in maintaining trace continuity, the alternate fill scheme does excel at generating traces at probably entry points, or region start points. As discussed in the context of fetch preconstruction, these points begin regions of code that will be encountered later in the course of normal execution. Probable entry points tend to start on basic block boundaries; also the manner in which the alternate fill scheme generates traces.

3.2 The Sliding Window Fill Mechanism & Fill Selection Table

The previous section identified two trace properties that the fill unit should take advantage of. The first is to maintain trace continuity when faced with a series of one or more *n*-constrained segments. The second is to identify probable entry points and generate traces based on these fetch addresses. A solution that satisfies this dual requirement involves a new scheme that incorporates the Sliding Window Fill Mechanim (SWFM) and the Fill Selection Table (FST). This section proceeds with a description of the FST followed by a description of the SWFM. A presentation and comparison of results is presented afterwards.

The concept of the fill selection table is fairly straightforward. Every cycle, the PC address that the core fetch mechanism encounters (following a trace cache miss) is identified as a probable entry point. These addresses are stored in the FST. An FST entry consists of an address tag, a valid bit and a counter. Each time a fetch address is encountered, the count value of the associated entry is incremented. The fill unit will also allocate or increment the count of a FST entry when an *n*-constrained trace is constructed and added to the trace cache. To provide multiple accesses per cycle, the FST can be implemented as an interleaved and/or multi-ported structure.

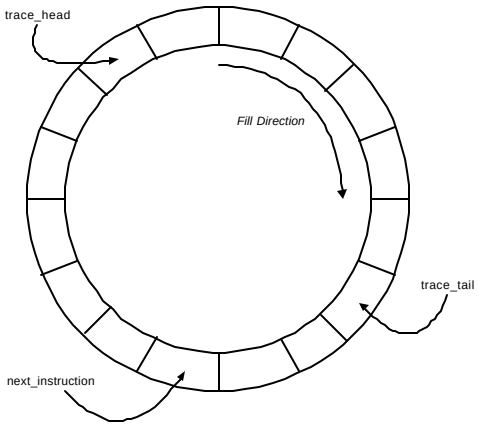

Figure 3: The Sliding Window Fill Mechanism

The Sliding Window Fill Mechanism that is paired with the FST is an extension of the alternate fill scheme examined in Section 2.3. The difference is that instead of "truncating-and-shifting" an entire basic block, single instructions are trimmed one at a time. The SWFM is implemented as a circular buffer, as shown in figure 3. Pointers are used to mark the current start of a

potential trace segment (trace_head) the final instruction of a potential trace segment (trace_tail) and the point at which retired instructions are added to the fill buffer (next_instruction).

When a retired instruction is added to the fill buffer the next_instruction pointer is incremented. At the same time, the potential trace segment bounded by the trace_head and trace_tail pointers is considered for addition to the trace cache (as described shortly). The trace_tail pointer is then adjusted according to the following table.

Table 2: Conditions for Adjusting the trace_tail Pointer

Fill Buffer Condition	Resultant trace_tail Value
`IsControlInstruction( trace_tail ) == FALSE`	*`Next-n-instr`*
`( IsConditionalBranch( trace_tail ) || IsDirectJump( trace_tail ) ) && ( IsConditionalBranch( trace_head ) || IsDirectJump( trace_head ) )`	*`Next-m-instr`*
`Trace-tail == trace_head`	*`Next-m-instr`*
`(otherwise)`	`trace_tail (no change)`

Next-n-instr is expressed as the following conditional statement:

```
trace_tail = ( trace_tail – trace_head ) < n - 1 ) ?
             trace_tail + 1 : trace_tail
```

Next-m-instr can be described with the following algorithm:

```
while ( ( trace_tail – trace_head ) < n - 1 ) {
    trace_tail++;

    if ( trace_tail == next_instruction )
        break;

    if ( isControlInstruction( trace_tail ) )
        break;
}
```

The last step in updating the state of the fill buffer is to increment the trace_head pointer.

From an implementation standpoint, this may seem to be a difficult task; independently the SWFM would generate as many traces as there are retired instructions! The value of the SWFM is realized when paired with the FST. Before filling a segment to the trace cache, a FST lookup using the trace ID is performed. If a corresponding entry exists, the count is compared with a defined threshold value. If this count meets the threshold, then the segment is added to the trace cache, and the FST entry is cleared. Otherwise, the state of the fill buffer is updated as described in the previous paragraph, effectively discarding the lead instruction. One final task of the fill buffer is to allocate or increment the count of an FST entry for any address that follows a n-constrained segment that gets added to the trace cache. In summary, the SWFM/FST combination identifies, constructs and fills traces starting with fetch IDs that have been previously encountered or those that immediately follow *n*-constrained traces.

4. SIMULATION AND RESULTS

4.1 Simulator Design

The SimpleScalar toolset was used as a starting point to verify the SWFM. The *sim-outorder* functional simulator, which modeled a superscalar, out-of-order issue pipelined CPU, was extended to accommodate trace cache schemes with a fair amount of complexity in a modular fashion. Using this approach, schemes ranging from the Rotenberg trace cache to more complex varieties could be implemented within the same uniform template. The primary goals associated with the design of the extension were flexibility and extensibility. Implementation in object-oriented C++ provided the means to fulfill the above, in addition to establishing a intuitive platform for further expansion. The reader is encouraged to reference [1] for further details on the simulator design.

4.2 Simulator Configuration

The *sim-outorder* parameters that were used in the simulation are given below in Table 3. Parameters that are not listed (such as latencies and penalties) were assigned default values by sim-outorder. Additional detail is provided in [1].

Simulation Parameter	*Value*
Decode/Issue/Commit Widths	16
IFQ Entries	128
RUU/LSQ Entries	512/256
Integer ALUs/Multiplication Units	16/4
FP ALUs/Multiplication Units	16/4
L1 I-Cache	
Size (lines)	1024
Line size	32
Associativity	Direct Mapped
Multiple Branch Predictor (MPAg)	
PAT entries	1024
PAT width (bits)	10
PHT entries	1024
BTB	
Entries	512
Associativity	4-way
RAS depth	8

Figure 3: Simulation Configuration

4.3 Benchmarks Used

The benchmarks chosen represent a variety of modern computational problems while possessing the benefits of relatively compact sizes and short run-times. The latter aspect was important to this study, as several trials were run to gain a perspective on a variety of schemes and organizations. Ten programs constituted the benchmark set. Table 4 presents a description of the benchmarks, the respective type of computation that each is characteristic of. A comprehensive overview of the benchmark programs and input sets is provided in [1].

Table 4: Benchmark Set Utilized

Bnchmrk Name	*Description*	*Computation Type*
roots	Finds root of a nonlinear equation via Bisection, Secant and Newton's methods.	Scientific Kernel
solve	Computes the root approx. for a polynomial using the bisection method.	Scientific Kernel
integ	Finds the approx. value for the definite integral of f(x) using Simpson's & Trapezoidal rule.	Scientific Kernel
lag	Lagrange interpolation of a set of data points, returns the value of function at x.	Scientific Kernel
matrix	Matrix multiplication	Scientific Kernel
gzip	File compression using Lempel-Ziv	Integer
djpeg	JPEG image decompression	Multimedia
cjpeg	JPEG image compression	Multimedia
fft	Performs Fast Fourier Transform on n random sinusoids sampling m times.	Embedded/DSP
inv_fft	Performs the inverse FFT	Embedded/DSP

4.3 Simulation Results

To observe how the Sliding Window Fill Mechanism affects the trace cache hit rate, a set of simulations were performed in which the fill selection threshold value was adjusted. The threshold value of the FST has a substantial effect on the number of unique traces added. Table 3 compares the number of traces added for each of the six FST threshold values. Figure 4 shows that that the best trace cache hit rate is obtained using a value of two or three.

Table 3: Number of Unique Traces Added

Bnchmrk	T=1	T=2	T=3	T=4	T=8	T=16
roots	25,725	4,828	2,121	1,828	837	327
solve	19,308	2,856	1,579	1,058	444	237
integ	11,480	1,968	1,013	708	303	144
lag	16,269	2,609	1,149	800	360	172
matrix	6,829	1,622	803	567	326	238
gzip	56,020	16,274	11,214	7,784	4,728	2,496
djpeg	327,289	25,265	15,770	11,488	5,297	2,344
cjpeg	303,101	39,720	24,436	18,703	8,666	4,500
fft	334,094	48,395	23,438	17,495	6,313	2,228
inv_fft	653,229	82,096	37590	26,158	10,167	3,068

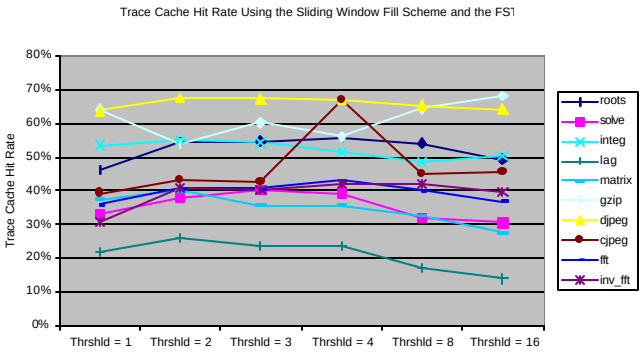

Figure 4: Hit Rate for SWFM using various FST Thresholds

Choosing a FST threshold of 2, the advantage of the Sliding Window Fill Mechanism can be shown through comparison with the Rotenberg scheme. Baring a few exceptions (gzip yielded consistently poor regardless of trace cache scheme), the hit rate and fetch bandwidths notably increased as shown in Figure 5 and 6. The average hit rate increase was 7% using the SWFM.

To conclude this study, two trace cache schemes that provided favorable results were combined: Branch Promotion and the Sliding Window Fill Mechanism. Intuitively, these schemes seem to compliment each other, as the SWFM excels at generating relevant traces, while branch promotion increases trace segment utilization by reducing the number of traces that are prematurely terminated by the *m*-constraint. The expectation that the merged scheme would increase trace cache hit rate along with the fetch held true, as figures 7 and 8 illustrate. For both metrics, the combined scheme outperformed branch promotion and the SWFM when implemented separately. The average hit rate increase over the Rotenberg scheme for this combined scheme was 19%. The fetch bandwidth improved, on average, 17% over the Rotenberg scheme.

5. CONCLUSION

This paper proposed a new fill unit scheme for trace cache that exploits trace continuity and identifies probable start regions to improve trace cache hit rate. Through simulation, it has been shown that this method increased hit rates by an average of 7% independently, and by an average of 19% when combined with branch promotion. The combined scheme also yielded a 17% increase in fetch bandwidth. These results demonstrate that the Sliding Window Fill Mechanism can contribute to the overall performance of a processor that utilizes an aggressive trace cache scheme.

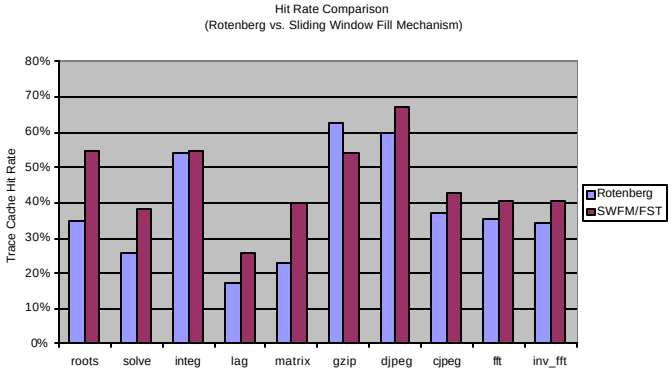

Figure 5: Trace Cache Hit Rate using the SWFM

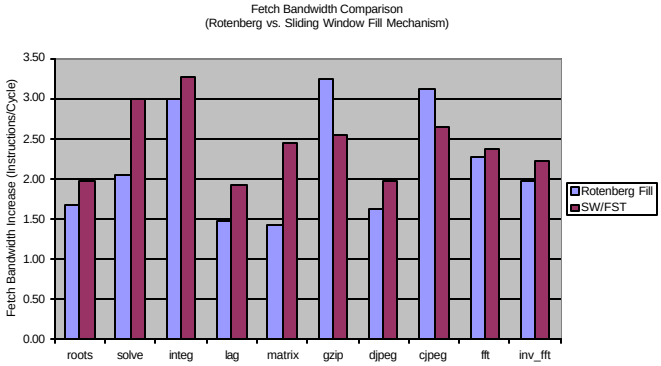

Figure 6: Fetch Bandwidth using the SWFM

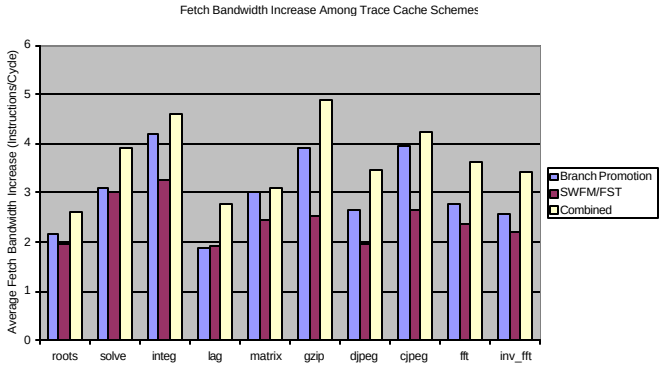

Figure 7: Fetch Bandwidth Increase for Combined Scheme

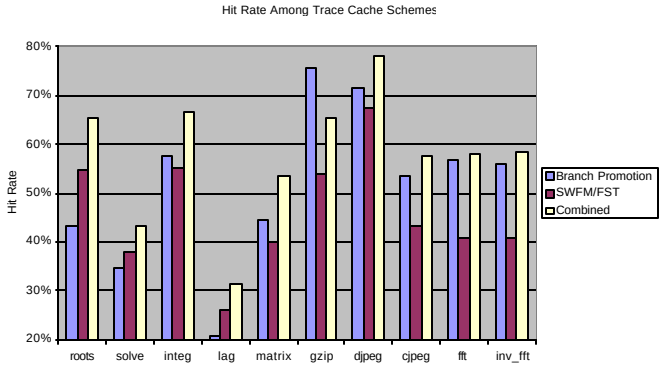

Figure 8: Hit Rate Increase for Combined Scheme

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge Roy S. Czernikowski and Ken W. Hsu for their valuable suggestions and feedback.

7. References

[1] E. M. Mulrane. "An Investigation of Trace Cache Organizations", MS Thesis, Rochester Institute of Technology, November 2002.

[2] E. Rotenberg et al. "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching". *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 24 – 34, 1996.

[3] S. J. Patel et al. "Improving Trace Cache Effectiveness with Branch Pomotion and Trace Packing". *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 262 -271, 1998.

[4] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line". US Patent number 5,381,533, Intel Corporation, 1994.

[5] A. Ramirez et al. "Trace Cache Redundancy: Red & Blue Traces". *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pp. 325 –333, 2000.

[6] Q. Jacobson and J. E. Smith. "Trace Preconstruction". *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 36 – 46, 2000.

[7] Q. Jacobson and J. E. Smith. "Trace Preconstruction". *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 36 – 46, 2000.