The Frequently Used Trace Cache

A Method For Improving Trace Cache Performance

Indraneel Datta, Marc Power and Eric Furbish

Abstract: *Trace caching provides a high-bandwidth instruction fetch mechanism for more effectively supplying a large execution core with a sufficient number of instructions. However, access time tradeoffs put a limit on how effective a large trace cache can be at increasing performance. With the addition of a smaller, more associative Frequently Used Trace Cache (FUTC) that employs a more judicious fill mechanism based on the usage frequency of trace cache lines, some of this access time can be hidden, yielding performance gains over a traditional monolithic trace cache.*

1 Introduction

With the increasing performance demands of today's applications, microprocessor designers are forced to create fetch mechanisms that can supply an ever-greater number of instructions per cycle to the execution core. Unfortunately, the traditional instruction cache hierarchy that has been employed thus far fails to scale to dispatch widths much larger than the size of a basic block, since the cache is designed to exploit spatial locality rather than trace locality. Recently, a different fetch mechanism called a trace cache has been proposed [1] to meet the needs of today's new processors. Trace caches, with the help of multiple branch predictors, store dynamic instruction traces rather than sequential areas of instruction memory and effectively widen fetch bandwidth by allowing one to take advantage of large multi-basic-block pools of instructions which are likely to be sequentially executable.

1.1 Motivation

These trace caches have been shown to improve machine performance [1]; additional testing illustrates that in the ideal case (with unit trace cache latency) performance scales with trace cache size (See Figure 1). As the number of available transistors on a chip increases towards 10^9, larger and larger on-chip trace caches will become ever more feasible; indeed, the area cost to include a 1MB trace cache, the point of diminishing size-performance returns revealed by our tests, will be relatively insignificant. However, there is a problem with large trace caches that overwhelms the previous space considerations: the increased access time required to access such large structures.

This issue is only exacerbated by recent trends in process technology scaling, as wire delays continue to become an increasingly large percentage of overall circuit delay, and additional simulation reveals the large trace caches that will be demanded by future microprocessors will require several cycles to access.

To verify this, one need only examine the figures below. Figure 1 represents the performance of a high execution bandwidth machine equipped with a variety of trace cache sizes, all of which can be accessed in a single cycle. In contrast, Figure 2

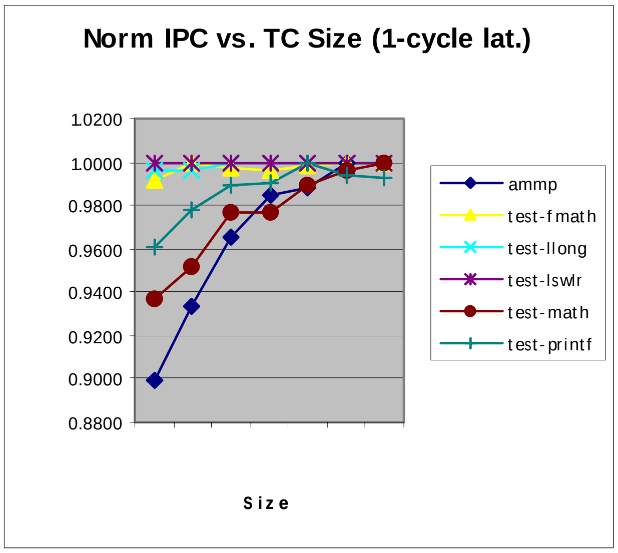


Figure 1

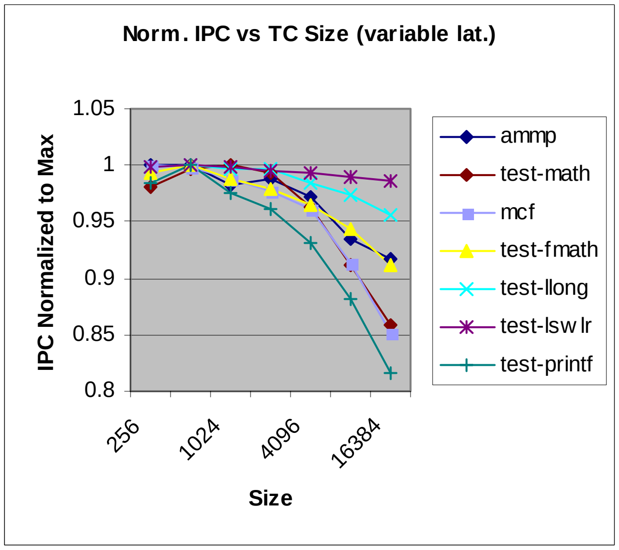


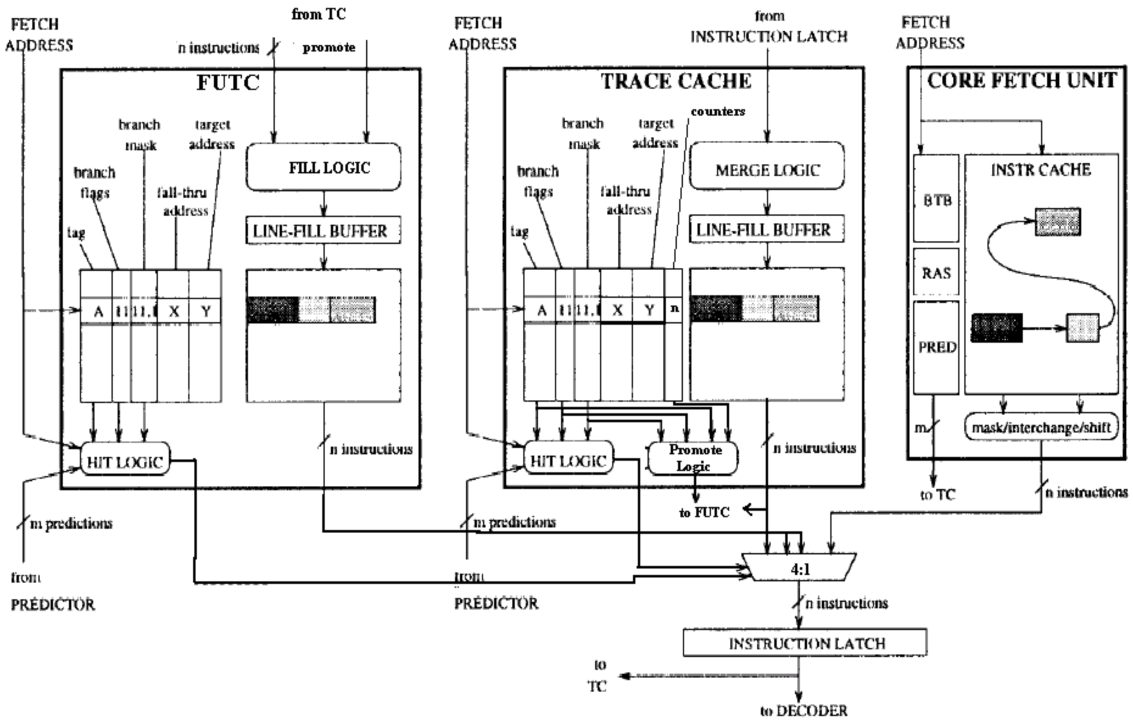
Figure 2

illustrates the behavior of the same machine modified to reflect the fact that trace caches of increasing size tend to have increasing latency. A comparison of the two quite unsurprisingly reveals that size-induced latency, an unavoidable part of any such large memory in a real machine, is clearly detrimental to performance. The ideal case shows the possibility of achieving roughly ten percent performance improvement if a method can be found for making a large trace cache more efficient.

The normal method used to combat this latency problem is to add another level of hierarchy to the cache structure. A small L1 cache with relatively small latency is used to store the most recently used data, while a larger L2 cache maintains more of the program in storage that is faster to access than memory. However, contention in the L1 cache, especially in large programs, can lead to decreased performance when frequently used lines are evicted and replaced by less frequently used lines. Thus, a method for judiciously filling the L1 cache with the lines that are most frequently used is warranted.

1.2 Concept

With this in mind, in an attempt to reclaim this performance lost to latency and contention, we propose a new structure called a Frequently Used Trace Cache (FUTC) designed to improve trace cache performance. The FUTC is a relatively small and associative trace cache that is accessed in parallel with the normal (L2) trace cache (Figure 3) and contains the lines most frequently used in the trace cache. A single saturating counter is added to each line in the L2 trace cache that is incremented each time that line is read. When that counter reaches a certain threshold, the line is promoted into the FUTC and the counter is left unchanged. This organization is similar to that of a traditional inclusive L1/L2 cache structure, except for the fact that writes into the L1(FUTC) occur only on trace cache read hits and are filtered by counters. We expect that the FUTC will improve performance over a traditional monolithic trace cache by providing a cache of frequently used lines that can be accessed in a single cycle. Further, we anticipate that the judicious fill method employed will improve the performance of the FUTC relative to a standard L1 trace cache of the same size.



(Figure 3: Fetch Pipeline augmented with FUTC; Figure modified from [1])

1.3 Overview of Paper

This paper will examine and evaluate the idea of a Frequently Used Trace Cache (FUTC) in detail. Section 2 contains a full overview of the FUTC architecture. Section 3 presents our experimental methodology, followed in Section 4 by the analysis of said experimentation. Finally, Section 5 concludes the paper.

2 Architecture

In Section 1.2, the idea of a Frequently Used Trace Cache (FUTC) was presented. This structure is effectively a small independent trace cache accessed in parallel with the general trace cache during instruction fetch. It draws its name from the fact that it is not filled on commit (or dispatch in architectures other than the one used for this study) as is the normal TC; rather, it is filled only with the most frequently used traces from the normal trace cache itself. The FUTC traces are not updated during commit since that would have required an extra write port (one for promotions, one for commits). Since the FUTC traces would have to be hit several times before getting to the FUTC, they should likely be correct in most cases.

This structure required a number of modifications to the fetch pipeline of the typical trace cache enabled fetch mechanism presented in [1]. First, each line in the trace cache is augmented with a saturating counter, representing how many times the line has been accessed since it has been written into the trace cache. This counter is zeroed when the line is written into the trace cache initially, and it is updated on TC read hits in the fetch stage of the pipeline. In addition, new logic and data paths must be added to the fetch pipeline to support the FUTC. These extra elements of the fetch mechanism consist of the FUTC itself, including its logic and storage requirements, as well as data paths between the trace cache and FUTC (to carry the promoted instructions and the PROMOTE signal). Of course, we must include the wire costs to route the address and branch prediction bits to the FUTC as well as the additional wires and larger multiplexer providing a path from the FUTC to the fetch queue on a FUTC hit.

All in all, the area and power costs of these additions are negligible, considering the small size of a practical FUTC accessible in a single cycle (roughly 8 – 64KB), the relative simplicity of the added logic, the small number of additional wires, and, of course, the high transistor counts of present and especially future processors. In a trace cache of 1MB (8192 lines), the counters would consume from 1-3KB, for counter sizes of one to eight.

In addition to its inherent hardware scalability, the FUTC provides a method of more effectively scaling performance to new processes, as it lifts the burden from large, monolithic trace caches and places it on a smaller, faster structure.

3 Experimental Methodology

To assess the effectiveness of the FUTC, the experiments were designed to address two specific issues. First, a large trace cache size whose performance lagged due to its large latency had to be picked for use with the later FUTC runs, as computing resources for running the simulations were limited. Second, the combination of FUTC size and FUTC promotion threshold that provided the best performance was sought. Once this combination was found, its performance was compared to the most reasonable design alternatives, the standard L1 trace cache (approximated with a FUTC of threshold 0) and a monolithic trace cache, to see whether the cost of implementing the FUTC would be worth the performance gain (if any). The standard L1 trace cache was approximated in the sense that its traces were only updated on a read hit in the L2 trace cache (as with the FUTC), instead of updating on all trace cache fills. As described in Section 2, this decision was made to avoid an extra write port on the FUTC, and because simulation showed performance was not degraded.

In picking the trace cache size that would be used for the later FUTC runs, two sets of inputs were used. First, the dependence of IPC on trace cache size was evaluated for trace caches with ideal single-cycle latencies (Figure 1). This data was used to determine how much performance was ideally available. Second, the same IPC dependence was generated for trace caches with non-ideal latencies that scaled with size (Figure 2, Table 1). Together, these two data points provided a trace cache size that would have approximately ideal performance if it were not dampened by its latency. The size that seemed most appropriate was 1MB, since it shows the last major gain in ideal performance with a significant degradation when latency is considered.

Unfortunately, lack of computing resources limited the number of benchmarks that could be run. However, a very representative set of SPEC2000 with reduced data sets was completed. The combination of sizes and thresholds was found using an exhaustive run for FUTC sizes varying from 8KB to 64KB (a range deemed suitable for single-cycle access) with thresholds from zero to

Size (KB)	*Latency (Cycles)*
32	1
64	1
128	2
256	3
512	5
1024	8
2048	12

Table 1: TC Latency vs. Size

Parameters For Trace Cache Runs	***Description***
`-bpred mgag -bpred:mgag 1 14 1 3`	Sets the branch predictor to multiple GAG mode, 1 BHR of width 14, XOR the address with the BHR, 3 predictions/cycle
`-cache:trace tc:<size>:16:3:1:l:n`	Defines a trace cache named tc of size <size>, 16 instructions and 3 branches per line, direct-mapped, LRU with normal matching
`-cache:tracelat <latency>`	Sets the latency of the trace cache
`-cache:futc futc:<size>:2:<thr>:l`	Defines a FUTC with name futc, size <size>, 2-way associative, threshold <thr>, LRU
Universal Parameters	***Description***
`-fetch:ifqsize 256`	Sets the ifetch queue size to 256
`-decode:width 32 -issue:width 32`	Sets the decode/issue width to 32
`-commit:width 32`	Sets commit width to 32
`-ruu:size 64`	Sets RUU size to 64
`-res:ialu 16 -res:imult 8`	Creates 16 integer ALUs and 8 integer multipliers
`-res:memport 8 -mem:lat 50 2`	Creates 8 memory ports with an initial memory latency of 50 cycles, with data returning every 2 cycles after that
`-res:fpalu 16 -res:fpmult 8`	Creates 16 FP ALUs and 8 FP multipliers
`-fetch:mplat 3`	Default misprediction latency
`-fetch:speed 1`	Default front/backend relative speed
`-bpred:ras 8 -bpred:btb 512 4`	Default RAS/BTB parameters
`-issue:inorder false -issue:wrongpath true`	Default OOO issue parameters
`-lsq:size 8 (See note)`	Default LSQ size
`-cache:dl1 dl1:128:32:4:l –cache:dl1lat 1`	Default L1 D$
`-cache:il1 il1:512:32:1:l –cache:il1lat 1`	Default L1 I$
`-cache:dl2 ul2:1024:64:4:l –cache:dl2lat 6 -cache:il2 dl2`	Default unified L2$
`-cache:flush false -cache:icompress false`	Default flushing/compression options
`-mem:width 8`	Default memory bus width configuration
`-tlb:itlb itlb:16:4096:4:l -tlb:dtlb dtlb:32:4096:4:l -tlb:lat 30`	Default TLB configuration

Note: This parameter is incorrect, but was discovered too late in simulation. However, since it is constant across all runs, we believe our results are still valid.

eight. The primary data points generated were for the *ammp*, *vpr* and *mcf* programs, with some points of secondary importance given for the small test programs that accompany the SimpleScalar tool set.

Additionally, the back ends for all simulations were set to approximate infinite execution resources as compared to the instruction fetch mechanism. This allows the full impact of the fetch method to be shown. The configuration included support for an instruction fetch queue of 256 entries, 16 integer ALUs, 8 integer multipliers, 8 memory ports, 16 FP ALUs and 8 FP multipliers. The branch predictor was a directly implemented large mGAG predictor[2] with a 14-wide BHR indexing a full 2^{14} entry PHT. The predictor generates three predictions per cycle, in accordance with the trace cache line limit of three branches. The trace cache held a maximum of 16 instructions/3 branches per line (the configuration found optimal in [1], and used a standard matching policy (i.e. every branch prediction must match the trace cache branch flags, or the line does not hit, unless the branch is at the end of the line).

The parameters used to run the modified version of sim-outorder are shown in Table 2. Parameters enclosed by "<>" were varied for each run. All others were constant. The -bpred:mgag, -cache:trace, -cache:tracelat and -cache:futc parameters are the only ones added to the original SimpleScalar parameter set.

4 Analysis

Figures 4&5 show the results of the FUTC size and threshold experiments. The immediate conclusion that can be drawn from the data is that an increasing threshold generally decreases performance. Though some cases fluctuate slightly, this trend is otherwise universal. There are several possible factors that may cause this. First, the programs available for simulation are generally of very small size. This means that contention in the L1 trace cache (which helped motivate the FUTC) does not affect performance to the same degree as with a

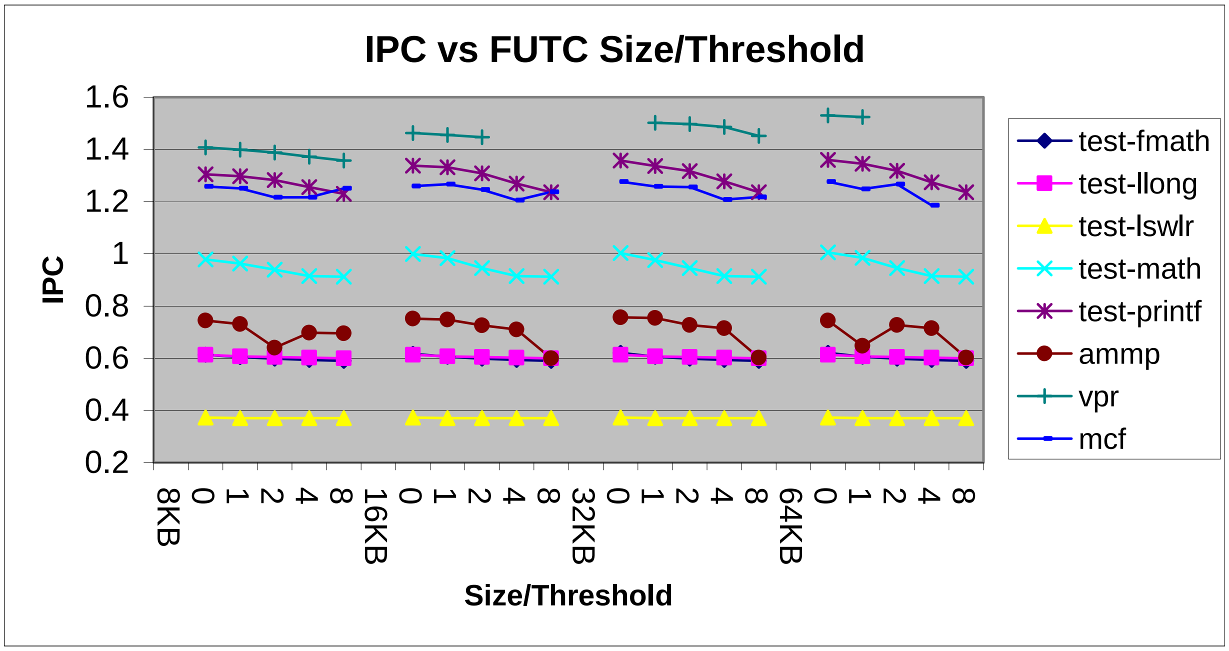


Figure 4

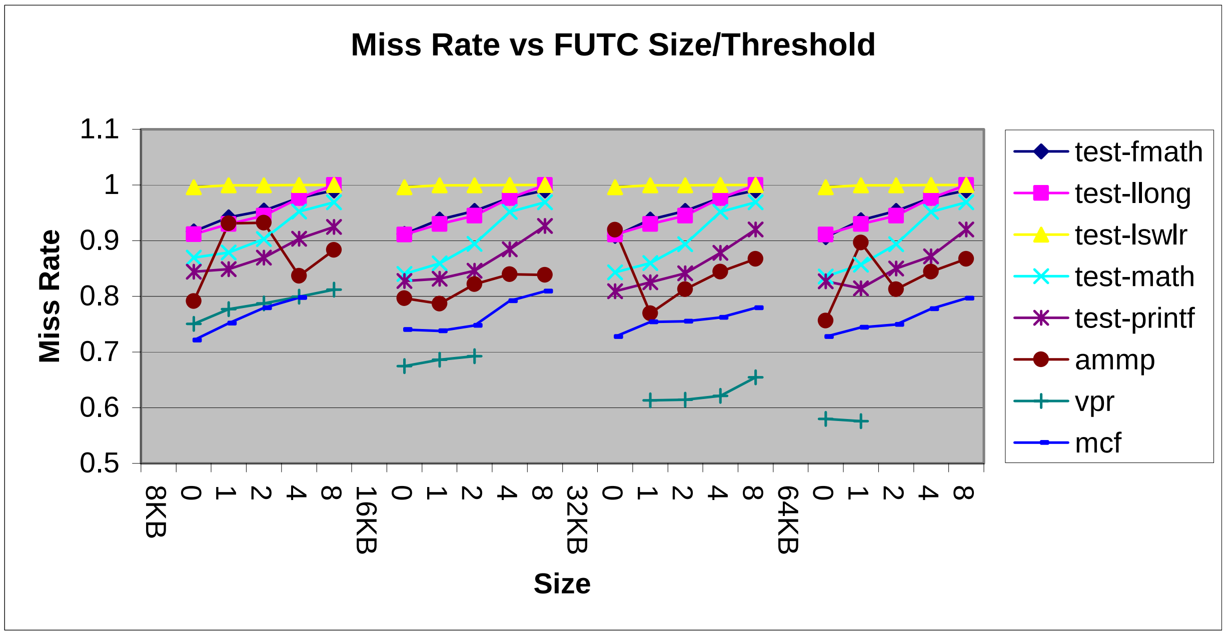


Figure 5

large program. Thus, increasing the promotion threshold merely makes the FUTC have a longer "warm-up time" than a standard L1.

Additionally, most of the simulated programs spend a lot of their time in localized loops that, once finished, tend not to be executed again for a long time, if ever. One possible effect of this property may be in its interplay with the algorithm for updating the traces in the FUTC. Poor interactions at this level may be causing at least part of the FUTC misses that lead to the poor performance. Recall that instructions are filled into the trace cache on commit, but not into the FUTC. The FUTC traces are updated when a promotion happens from the trace cache. Hence, if a trace inside one of these localized loops is placed into the FUTC and not replaced, and the actual program trace changes, then the FUTC trace for that address will be incorrect. Assuming the branch predictor begins correctly predicting the trace, misses will start to occur in the FUTC; indeed, Figure 5 shows this to occur. Further simulation would be required to decide whether this update algorithm degrades performance. If it were found that updating the FUTC entries on TC commits to lines with similar tags is beneficial, the benefit of implementing it would need to be weighed against the cost of the additional FUTC write port needed to do it efficiently.

However, whatever the effect of the exact mechanics of filling, the high miss rates are, for the most part, obviously attributable to the higher promotion threshold itself causing fewer valid traces to be present in the FUTC at any given time. When a localized loop is present in the FUTC and the program moves on, it must begin re-filling the FUTC with the new traces that it is executing. The counters may prevent this filling from happening quickly, and if the program moves on it will suffer misses while it is constantly forced to warm up the FUTC.

In any case, regardless of the reasons for the poor performance of the FUTC-augmented

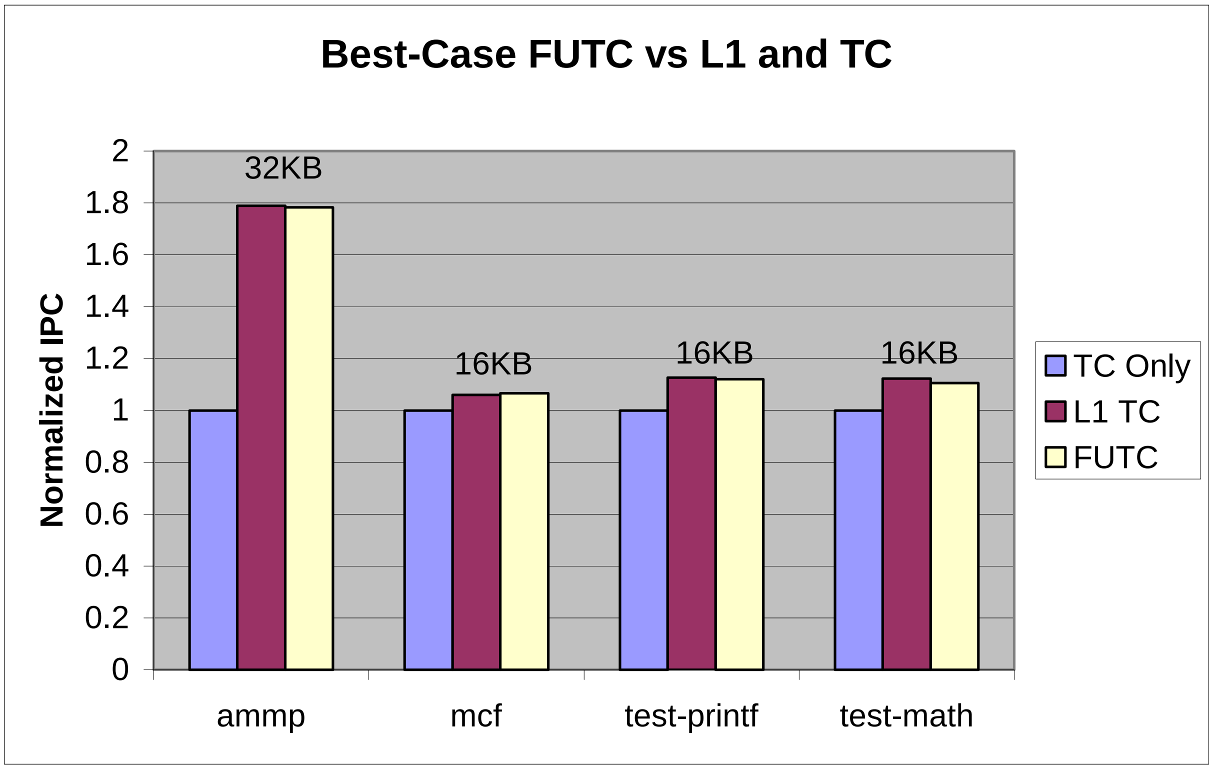


Figure 6

machine, it seems clear that while the FUTC did provide some benefit over a machine with only a simple trace cache mechanism, it did not generally out-perform a fetch mechanism equipped with a "classical" L1/L2. Figure 6 compares normal TC performance against that of the FUTC- and L1-augmented systems. In all cases, both setups containing an auxiliary trace cache structure outperformed machines with a general TC fetch structure, gains which seem *clearly* due to the presence of any small, fast auxiliary TC at all. More importantly, however, this graph illustrates that in general there are no gains to be had by using a FUTC over an L1 TC. The data for FUTC runs represents the optimal performance of the FUTC for a given application (while the TC+L1 data merely come from configurations with corresponding second-level structure sizes). Even in this situation, in basically all cases the L1 outperforms the FUTC. Furthermore, if one examines Figure 4, the relative superiority of the L1 is even more clearly displayed – the best case L1 configuration always achieves greater performance on a given benchmark than the best case FUTC configuration.

Despite the relatively poor performance exhibited by the FUTC on this set of benchmarks, there may be some hope for it yet. It seems likely that the FUTC would exhibit different characteristics on extremely large programs, or more precisely programs with large or varied instruction memory working sets, where contention would dominate the L1/FUTC miss rate. However, unavailability of such programs and extremely limited simulation resources rendered testing this hypothesis impossible. Similarly, trace line matching policies such as partial matching and inactive issue drastically increase the trace cache hit rate, making L1 contention a much larger bottleneck and perhaps paving the way for FUTC-based performance improvements. For now, however, this is only conjecture, and must be borne out by experimentation before given overmuch weight.

5 Conclusions & Future Work

As alluded to in the previous section, the improvement in IPC presented by both the FUTC and the L1 trace cache over the base configuration clearly illustrates the presence of performance gains to be had with some form of small fast auxiliary trace cache structure. This bodes well for the general issue of overcoming the latency issues caused by increasing trace cache sizes and process scaling (or lack thereof). However, as a specific means of ameliorating TC latency issues, the FUTC clearly failed the litmus test. In failing to outstrip a simple additional L1 trace cache, the FUTC was shown to be only useful inasmuch as it provides fast auxiliary TC storage; indeed, it seems to be little more than a slow-to-warm-up L1 trace cache. Thus, modulo later testing with larger programs and other matching schemes, the FUTC should be abandoned in favor of other forms of fast TC auxiliary. Beyond the effective simple L1 trace caches, there are a host of other caching optimizations that can and should be tried. After all, if one “classical” optimization shows positive results, why might another not be even more favorable? At the very least, it is clear that the problem of hiding increasing trace cache latency is far from closed.

REFERENCES

[1] E. Rotenbert, S. Bennett, J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. Tech Report 1310, CS Dept., Univ. of Wisc.-Madison, 1996

[2] T.-Y. Yeh, D.T. Marr, and Y.N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache, *7th Intl. Conf. On Supercomputing*, pp. 67-76, July 1993