Minimizing Area Cost of On-chip Cache Memories by Caching Address Tags *

Hong Wang, Tong Sun, and Qing Yang
Dept. of Ele. & Comp. Engineering
University of Rhode Island
Kingston, RI 02881
e-mail: {hwang,tsun,qyang}@ele.uri.edu

Abstract

This paper presents a technique for minimizing chip-area cost of implementing an on-chip cache memory of microprocessors. The main idea of the technique is *Caching Address Tags*, or *CAT cache* for short. The *CAT* cache exploits locality property that exists among addresses of memory references. By keeping only a limited number of distinct tags of cached data rather than having as many tags as cache lines, the *CAT* cache can reduce the cost of implementing tag memory by an order of magnitude without noticeable performance difference from ordinary caches. Therefore, *CAT* represents another level of caching for cache memories. Simulation experiments are carried out to evaluate performance of *CAT* cache as compared to existing caches. Performance results of SPEC92 programs show that the *CAT* cache with only a few tag entries performs as well as ordinary caches while chip-area saving is significant. Such area saving will increase as the address space of a processor increases. By allocating the saved chip area for larger cache capacity, or more powerful functional units, *CAT* is expected to have a great impact on overall system performance.

1 Introduction

Rapid advances in VLSI technology have made it possible to have powerful single-chip processors such as superscalar processors, superpipeline processors and vector processors [1, 2, 3]. Such single chip processors typically allocate about half of the chip area for on-chip memories such as register files, TLB and caches. On-chip caches are the most important part of memory hierarchy in today's single-chip processors. While these on-chip caches significantly improve processing performance, they also occupy a large portion of the scarce resource of chip area [4, 5, 6]. Since functional units, pipelines, instruction issue logic, registers, TLB, caches etc. all compete for this limited silicon real estate on a chip, any area savings resulting from a cost-effective on-chip memory design can have a significant impact on overall system performance.

An on-chip cache memory generally consists of two relatively independent areas, one for data, called *data area*, and the other for address tags and status bits, referred to as *tag area* [4]. The data area stores actual data needed by program executions, while the tag area is the storage overhead needed to determine whether a cached data line is indeed the data being referenced. There is one to

*This research is supported in part by National Science Foundation under Grants No. MIP-9208041, MIP-9505601

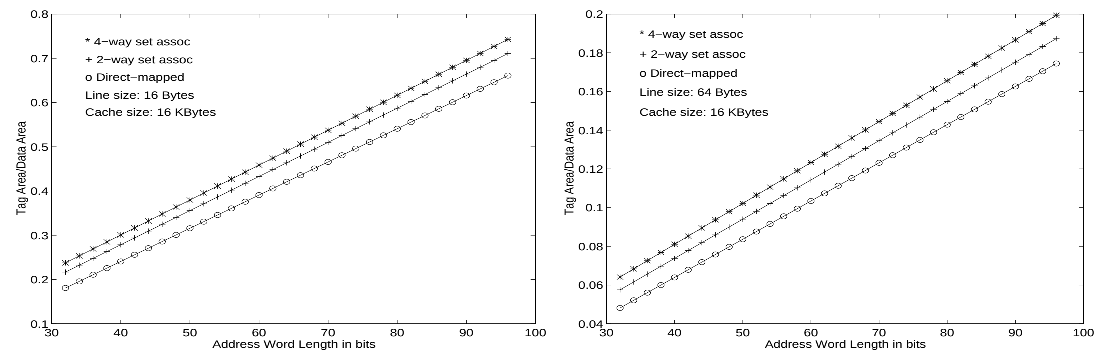

Figure 1: The ratio of tag area to data area for an on-chip cache.

one correspondence between the data area and the tag area, that is, each cache line in the data area has one and only one address tag corresponding to it in the tag area. For good cache performance, we would like to have a large data area since the cache hit ratio increases with the increase of cache size [7, 8, 9]. However, increasing data area also results in an increase in the tag area. Such an increase in the tag area will be magnified in the future as technology advances. Figure 1 shows the ratios of tag area to data area in a 16K-byte cache with two different cache line sizes (16 bytes and 64 bytes) as a function of address word width based on the chip area model presented by Mulder, Quach and Flynn [4]. It is clear from this figure that the tag area has already taken a notable fraction of space as compared to data area in today's computers. It will take more area as the memory address space increases. Hennessy and Jouppi [10] predicted that memory address space grows about 1 bit per year. We have already seen commercial microprocessors with 64-bit addresses [1]. The growth of memory space driven by application environment continues and may even accelerate [10]. Therefore, we have reached a point where it is desirable to redesign the cache memory to stop the growth of the tag cost, especially for on-chip caches.

In order to minimize the cost of implementing the tag area of an on-chip cache, we took a closer look at the locality property of memory references. Because of the spatial and temporal locality property of references, address tags of cached data are clustered at a given time frame of program execution. As a result, many tags are identical during a period of time. In other words, the effective working set of unique tags is much smaller than the working set of data references. We observed this phenomenon through extensive simulation experiments by taking snapshots of cached data and corresponding tags. Looking at any of these snapshots, we found that most tags are identical. Keeping track of all the redundant tags in an on-chip memory may not be necessary and is clearly expensive. Therefore, we propose a new technique referred to as *CAT—Caching Address Tags*. The main idea of our technique is very simple. Instead of storing all the tags corresponding to cached data lines, we use a very small cache memory, called *tag cache*, to store only distinct tags. The *tag cache* represents another level of caching inside cache memory to exploit the locality property of addresses of memory references.

Trace-driven and execution-driven simulation experiments are carried out to study the potential performance of the *CAT* cache design. Cache performances of eighteen SPEC92 benchmark programs are measured through simulation experiments. It is shown that the *CAT* cache with 16 or 32 tag entries performs as well as ordinary direct-mapped, write-through caches with 1K or 2K tags. The absolute difference of miss ratio between the *CAT* cache and ordinary caches is generally less than 0.01% for all benchmark programs considered. We have also carried out tag cost analysis based on the chip area model presented by Mulder, Quach and Flynn [4]. The chip-area savings resulting from the *CAT* are shown to be significant.

The paper is organized as follows. The next section discusses the prior existing research as related to our work. Section 3 presents the *CAT* design and related implementation issues. In Section 4, simulation environments are described. Performance evaluations are presented in Section 5 followed by area cost analyses in Section 6. We conclude our paper in Section 7.

2 Related Work

As we know, an address tag is the high order part of memory address associated with a cache line in the data area. For a given cache size, increasing the cache line size reduces the number of tags and hence the tag cost because of a reduced number of lines. However, extensive research in cache memories has shown that increasing cache line size beyond certain point results in performance penalties. First of all, the cache line is the basic data unit based on which data transfer between cache and the main memory takes place. For the same cache miss ratio, large cache lines result in high traffic between memory and processor [11]. Large cache line sizes may also cause cache pollution [12], because some data in a large cache line may never be used after being loaded into the cache. Experiments have shown that most applications prefer small to medium cache line sizes [7], that is, for many applications, the lowest miss ratio occurs at small to medium cache line sizes such as line sizes between 16 bytes and 64 bytes. Furthermore, large cache line size may also cause *false sharing* that is one of the main sources of performance degradation in cache-coherent multiprocessors [13, 14, 15, 16].

An alternative cache design that can reduce tag cost is *sectored cache* that has been implemented on several commercial processors [17, 3]. In a sectored cache, several consecutive memory blocks are grouped together to form a sector in the cache that share a same tag. While sectored cache can reduce the tag cost, it generally shows higher miss ratio than nonsectored caches. Seznec has proposed an interesting technique called *decoupled sector cache* [18] which not only reduces tag cost but also improves the performance of sectored cache. While Seznec's work aims at decoupling the tags from data in a sector of a sectored cache, the objective of our work is to exploit the locality property of addresses of memory references. In order to achieve reasonable cache performance, a sectored cache or decoupled sectored cache usually requires large cache size. Therefore, sectored caches are used mainly for second level caches having a large cache capacity. Furthermore, in a decoupled sectored cache, several tags are statically allocated to a sector of consecutive cache lines, while the *CAT* cache carries out dynamic tag mergings to effectively exploit the locality property of addresses of memory references, as will be evidenced shortly.

Locality property of memory references is well known to all computer architects. This locality concept is generally understood as data locality that means data referenced by CPU show spatial and temporal locality. *Address locality* is a direct consequence of data locality. By address locality, we mean that the addresses used to reference data are clustered in a given time frame of execution. In other words, only a small portion of an address word changes frequently while the remaining portion stays unchanged during certain time of execution.

The existence of address locality of memory references has been known for a long time [19, 20]. Very recently, Farrens and Park presented a very nice technique, called *dynamic base register caching* [21] to reduce address bus width by exploiting address localities. Instead of transferring complete addresses over a bus, they proposed to transfer a portion of a memory address while the other portion is cached at both memory and processor sides. Their technique has proven to be successful by trace-driven simulation experiments. However, to the best of our knowledge, no one has ever tried to exploit the *address locality* property of memory references for cost effective on-chip cache designs.

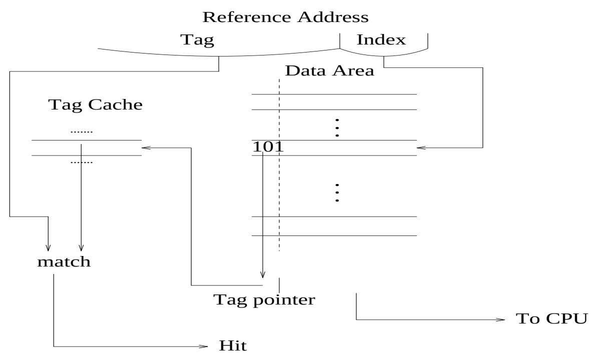

Figure 2: Block diagram of an implementation of *CAT* cache.

3 The *CAT* Cache

Figure 2 shows the schematic diagram of one possible implementation of *CAT* cache. It consists of *data area*, *tag pointers* and *tag cache*. The organization of the *data area* is the same as the existing cache organizations. It stores cached data lines. The *tag pointers* associated with cache lines serve as linkers to link a tag in the *tag cache* to cache lines in the *data area*. The *tag cache* holds distinct address tags of cached data lines. The size of *tag cache* is expected to be very small in the range between 8 to 32 entries. For each cached data line, there is only one tag in the *tag cache* that corresponds to it. However, a tag in the *tag cache* may correspond to one or several cached data lines. Such one-to-many relationship is realized by tag merging to be described shortly. To make our description concise, we consider only write-through caches. Most on-chip caches are write-through and the advantages of write-through caches are discussed in [22]. In addition, we assume speculative use of cache data is supported where data are made available to CPU even before the system knows whether a hit will occur [23]. In the following discussions, operations and several design issues of the *CAT* cache are presented.

3.1 Operations in *CAT* Cache

Cache Hits: Upon a memory reference, the index field of the reference's address locates the data item in the data area and its corresponding *tag pointer*. The data located by the cache index of reference's address in the cache is immediately loaded into a register buffer and made available to the CPU. The *tag pointer* activated by the reference's address points at one tag in the *tag cache*. When the tag part of the reference's address is available, usually after the address translation (TLB look up), it is compared with the tag in the *tag cache* pointed at by the *tag pointer*. If a match is found, a *cache hit* occurs. The hit time of the *CAT* cache is the same as that in an ordinary cache. If the address translation of the tag part is not necessary, the tag cache can be accessed in parallel with data access as discussed in Section 3.2.2

Cache Misses: When the tag in a reference's address does not match the tag pointed at by the *tag pointer* activated by the index field of the reference address, a *cache miss* occurs. Memory access operation is then initiated to bring the missed data into the cache from the off-chip L2 cache or the memory. During the time when the memory operation is in progress, the tag of the reference's address is again compared with all tags in the tag cache to see if there is any match. A match indicates that the currently referenced data item shares the same tag with one or several other cache lines already in the cache. The tag of the current reference does not need to be stored in the tag cache. *Tag merging* operations are performed by writing the location of the matched tag in the tag cache into the *tag pointer bits* associated with the missed cache line. This write operation should be performed at the same time when the missed data arrives at the cache.

If the tag in the reference's address does not match any tag in the tag cache, a *tag cache miss* occurs. A new entry in the tag cache should be allocated to the tag of the current reference. If there is a free entry in the tag cache, i.e. the tag cache is not fully occupied, then the miss is essentially a cold start miss and the tag is stored in the free entry. Otherwise, a replacement algorithm (to be discussed shortly) is initiated on the *tag cache* to replace a tag in order to make a room for the new tag. In any case, the *tag cache* location where the new tag is stored is copied to the *tag pointer* field addressed by the index field of the current address. As a result, whenever a new cache line is brought into the cache, the *tag pointer* field of the new cache line is updated. The content of this *tag pointer* is the *tag cache* location for the tag corresponding to the new cache line. All these operations within the *CAT* cache are carried out concurrently with the memory access for the missed data line and they should take less time than the memory access. Therefore, the cache miss penalty incurred on the *CAT* cache remains unchanged as compared to an ordinary cache.

In case of nonblocking cache, the cache supplies a "hit under miss" to allow CPU to continue out of order execution while missing data is being fetched. The complexity of the cache controller is significantly increased as there are multiple operations going on in the cache. Such operations include searching the tag cache and updating the tag pointer in the data array. At the same time, additional cache accesses may be issued by the CPU. The increased hardware complexity may also increase the average cache access time. It is possible but needs further study to design a nonblocking *CAT* cache without increasing miss penalty.

Tag Replacements: If the *tag cache* is fully occupied when a *tag cache miss* occurs, a tag replacement operation needs to be performed. There are several possible policies to replace a tag from the tag cache. The simplest and well-known replacement algorithm is LRU, that is, replacing the *least recently used* tag. A replaced tag will also be referred to as *victim tag.* It is important to note that in the tag cache, unlike in a regular cache, *tag replacement must be accomplished by invalidating all cache lines that are associated with the victim tag.* This invalidation is necessary to ensure program correctness as evidenced in the following example.

Example 3.1 Consider a sequence of memory references to the following memory addresses (not necessarily consecutive references)

tag	index	block name	Cache Contents	Explanation
0x20	0x2	A	x_1	Block A contains data x_1
...	...	...		
0x10	0x1	B	x_2	A's tag, 0x20, is repaced by B's tag, 0x10
0x10	0x2	C	x_3,	New data block C shares the same tag as B, also the same index as A. x_1 is taken wrongly by a false hit if A was not invalidated when A's tag was replaced.

Suppose that a cache miss occurs when CPU references block B and the tag cache is full. Assume that tag (0x20) (matching the tag of block A) is the victim tag being replaced in order to make a room for tag (0x10) of block B. If block A, i.e., line 0x2, were not invalidated accordingly, then upon the next cache access for data C which has the same cache index as A but different tag, CPU would falsely identify a hit in the cache and get an erroneous data (block A) for C. This scenario is called **false hit** because it would result in logical errors in programs.

From this example, we can see the necessity for the aforementioned invalidations of all the cache lines associated with the tag being replaced from the *tag cache.* Clearly, invalidation is a direct consequence of *CAT* cache's one-to-many mapping between *tags* and *cache lines* for maintaining coherence of cached addresses.

3.2 Implementation Issues

In this subsection we discuss several implementation issues of the *CAT* cache design including how to invalidate cache lines associated with a victim tag, optimal tag replacement and different *CAT* cache configurations.

When a tag is purged from the tag cache upon a tag miss, hardware to invalidate the associated cache lines is required. In a write-through cache, it is not necessary to write data line back into the next level memory for invalidated cache lines. Instead, only the *valid bits* of the affected cache lines need to be cleared. The invalidations can be done simply by comparing the binary representation of the location of the victim tag to the *tag pointer*. All the pointers that match the input clear the corresponding valid bit. Since the tag cache is small between 8 and 32 entries, the number of bits in a pointer is between 3 and 5. The simple 9 transistor CAM cell [24] can be used to clear the corresponding valid bits. Because the invalidations are performed only on a cache miss, the cost of such pointer memory is small as shown in our cost analysis presented later in this paper. For write-back caches, the problem is more complicated because write-back of changed cache lines is necessary. Further study is needed to evaluate design tradeoffs of write-back *CAT* caches.

Least Counter Tag Replacement Policy: Since invalidations of affected cache lines are necessary to guarantee program correctness in the *CAT* cache, it is possible that some frequently used data items are invalidated unfairly. The resulting cache thrashing will have adverse effects on cache performance. For example, in several data intensive programs of the floating point SPEC92 benchmark suite, multiple data array variables coexist with a smaller number of scalar variables in many loops. For a given cache configuration, the data area may not be large enough to hold all data variables. In a *CAT* cache it would be ideal to replace a tag for a scalar variable rather than an array variable because the tag for the array variable may correspond to a large number of array elements.

To minimize cache thrashing, a number of techniques may be considered. A counter with a few bits can be introduced for each tag entry in the tag cache. The counter is incremented every time when a tag merging operation is performed. The counter is decremented whenever a data line whose *tag pointer* points at the tag of interest is replaced from data area. Once a tag becomes a *victim tag*, the counter is reset to 0. The counter keeps track of the number of cache lines that share this tag. Therefore, the counter value of a tag, as the number of cached data lines that are associated with the tag, reflects the relative significance of this tag.

Whenever there is a *tag cache miss*, we replace a tag entry that has minimal counter value. Such tag replacement policy is referred to as *least counter replacement* policy (*LCR*). It can be shown that using tag counter alone without considering temporal locality (as LRU does) cannot be effective. For example, any new tag that has just been placed in the tag cache must have the least counter value 1. It is therefore liable to be replaced upon next tag cache miss. A more judicious least counter replacement policy should be integrated with LRU replacement policy. For example, the least counter comparison should only be applied to the less frequently used tags. In a simple strategy, a *most recently touched* bit can be associated with each tag cache entry, this bit will be set for the tag that is shared by a latest reference address. The entry with its *most recently touched* bit set to *true* should be excluded from the candidate set of victim tags.

There is a trade-off between the cost of implementing such counters and the performance gains. To implement the counter precisely, we would need $log_2(number\ of\ cache\ lines)$ bits for the counter.

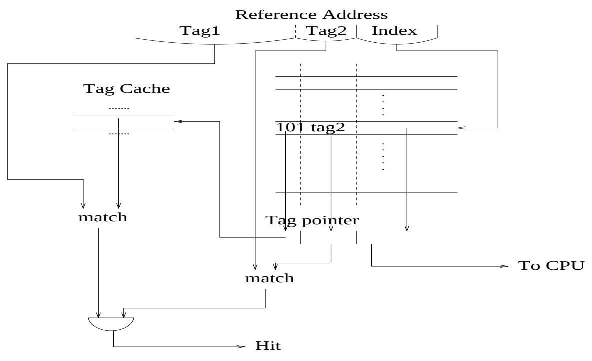

Figure 3: Block diagram of an implementation of Partial CAT cache.

However, in practice a couple of bits may be sufficient to approximate the relative significance of the tags in the tag cache. In the worst case when counters for all tag cache entries reach maximum, the policy is reduced to pure LRU.

3.2.1 Partial CAT Cache

Through our experiments (to be described in Sections 4 and 5), we observed that a small part of tag changes frequently during program executions while the remaining part keeps unchanged. In order to reduce the tag cache size and exploit address locality further, we can consider to partition the tag field into two parts: high order bits and low order bits. Only the high order bits in tag field are cached in the tag cache, while the low order bits are saved as tag arrays in an ordinary cache organization, i.e. there is a one-to-one correspondence between the tag's low order part and the cache lines. We call this organization *partial CAT* cache, or *ParCAT* for short. Figure 3 shows the block diagram of the *ParCAT* design.

As shown in Figure 3, the tag field of an address is partitioned into two parts, $Tag1$ and $Tag2$. The $Tag2$ field of each cached data line is stored in its original form associated with the cache line. We expect a great deal of information contents in the $Tag2$ field for cached data but a minimum amount of information contents in the $Tag1$ field. As a result, the tag cache size is reduced even further. However, whether such tag partition will be beneficial needs further validation by experiments since the address locality is highly program dependent. Our simulation results show that all benchmark programs considered benefit from such tag partition. Therefore, the number of tag entries in the tag cache can further be reduced to retain the same cache performance.

3.2.2 Parallel CAT Cache

In the CAT cache design depicted in Figure 2, tag cache is searched sequentially after accessing the data array. Therefore, if the CPU lacks the ability to speculatively use data in the cache line that is located by the cache index, CAT cache will incur one more step to determine cache hit/miss than an ordinary direct-mapped cache.

Figure 4 shows an alternative organization of CAT cache, *parallel CAT*. In this CAT cache, tag cache and data array are accessed in parallel. That is, tag field of the given reference address is used to find a matched tag in the tag cache while the cache index field is used to locate the cache line of interest and its tag pointer. Associative search of the tag cache for a match and the decoding of the index for data array are done in parallel. As a result of these two parallel operations, the address of the matched tag in the tag cache and the tag pointer associated with the indexed data are available. The tag address and the tag pointer are then compared to determine a cache hit or

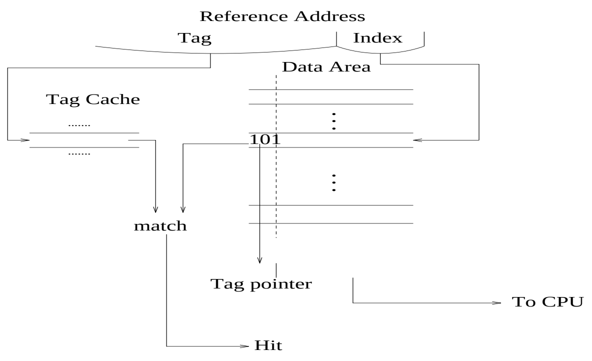

Figure 4: Block diagram of an implementation of Parallel *CAT* cache.

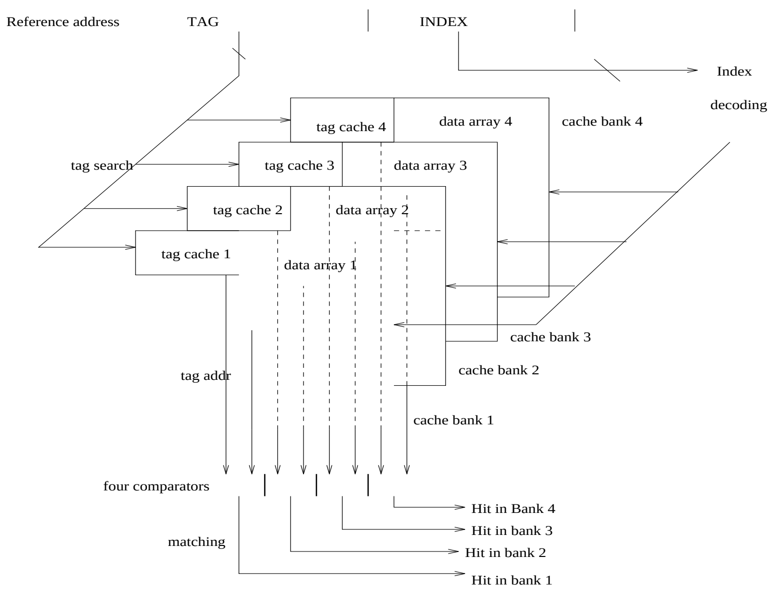

Figure 5: Block diagram of a 4-way set-associative *CAT* cache, n-tag-n-cat configuration.

miss. Since tag cache has very few entries (8-32) and is organized as CAM array, tag cache search time will be hidden by the time for cache index decoding and valid bit checking on data array. Therefore, parallel *CAT* cache has same critical path length as an ordinary direct-mapped cache, no matter whether the CPU can speculatively use data in a cache line before tag comparison or not.

3.2.3 Set-associative *CAT* Cache

The CAT cache described previously can easily be extended to n-way set-associative cache. In this subsection, we will discuss how to extend the CAT into set-associative cache memories.

An n-way set-associative cache usually consists of n identical cache banks, each of which is organized as an ordinary direct-mapped cache [23]. Corresponding to a given cache index, there are n distinct physical cache lines in n distinct cache banks. To implement the CAT cache, at one end of the spectrum, each cache bank in the n-way set-associative cache can be organized as a direct-mapped CAT cache. As a result, there are n identically organized direct-mapped CAT caches in an n-way set-associative CAT cache as shown in Figure 5. For each memory request, all n cache banks are accessed in parallel. The index field will activate n cache lines while the tag field is sent to the n tag caches to find, in the corresponding tag cache, a matched tag which also

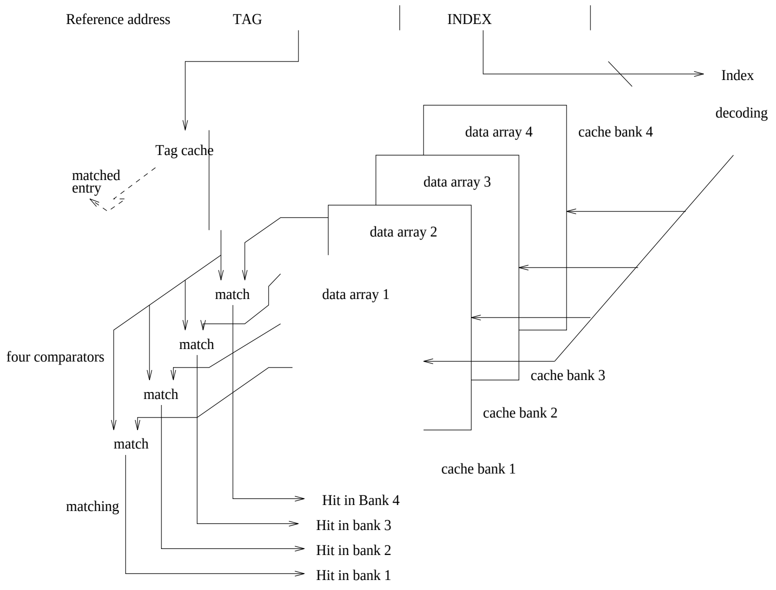

Figure 6: Block diagram of a 4-way set-associative *CAT* cache, 1-tag-n-cat configuration.

matches the tag pointer of the cache line. If such matching is found, a hit occurs. Otherwise, we have a miss. Note that at most one valid matching line can be found in the parallel search among n activated cache lines. In case of a miss, the LRU (or other replacement) algorithm is performed to select a cache bank where the activated cache line is to be replaced. Tag merging is performed only within this cache bank and the tag pointer field of the new data will be updated. Within each direct-mapped CAT cache, a tag within the tag array is "*vertically* unique" and can correspond to multiple lines in the data array within that cache bank. Since the association between the tag cache and the data array is confined within a cache bank for one specific way of associativity, replacing a tag may invalidate multiple lines within the cache bank, but it will not and should not invalidate any line in other $(n-1)$ banks.

On the other end of the spectrum, the association between tag array and data array can be defined globally across the boundary of cache banks. That is, a single global tag array can be defined in which each tag entry can be associated with multiple data lines in different data banks as shown in Figure 6. The cache access procedure is similar to the one described above except that the tag of a reference address is sent to the global tag cache. The address of matched tag will be compared with all n tag pointers from the n cache banks. Certainly, at most one match can be found. With this design, however, replacing one tag from the tag array will affect multiple lines in multiple cache banks. Potentially, this could incur ripple effect of invalidation to replace otherwise valid lines across various ways. As the design scales from multiple tag cache to single global tag cache, the area cost curves down accordingly, though the performance in terms of miss ratio may scale inversely due to potential over invalidation across the boundary of multiple cache banks.

4 Simulation Methodology

In this section we describe the simulation experiments that were conducted to quantitatively evaluate the performance potential of our *CAT* cache design. For the purpose of comparison with existing cache organizations, we define a *baseline cache* which is an ordinary direct-mapped cache. The *baseline cache* differs from the *CAT* cache only in tag memory organizations. The *baseline* *cache* has as many tags as the number of cache lines while the *CAT* cache has a small *tag cache*. The data area of both the *CAT* cache and the *baseline cache* are the same.

For benchmark programs, we considered the widely accepted SPEC92 benchmark suite due to their realistic nature and acceptable portability. In order to provide a reasonably comprehensive assessments of the *CAT* cache, we have simulated the entire SPEC92 suite except for Ora and Spice. The reason why we omit these two programs is that the cache miss ratio of program Ora on both the baseline cache and the *CAT* cache is zero for cache sizes exceeding 8K bytes, and the trace size of Spice is excessively large [7]. The SPEC92 contains both C programs and Fortran programs. To speed up our simulation process, we carried out two sets of simulations in parallel. One is trace driven simulation and the other is execution driven simulation.

4.1 Trace Driven Simulation

Our traces were collected on the fly using Pixie utility on DECStation 5000/200 which uses MIPS R3000 RISC processor running DEC Ultrix 4.2a. For comparative study, two simulators were used for each benchmark program: one simulating a baseline cache configuration; the other simulating a *CAT* cache configuration. The statistics collected in the two configurations provide insights to (1) the ability for tag-cache to capture the address locality, and (2) the *CAT* cache's overall performance affinity to the performance of the baseline cache.

In conformance to SPEC92's evaluation requirements, especially to retain the originally specified compiler optimization levels for benchmark programs, we generated the benchmark executables using the makefile script (i.e. M.dec_risc) from the original SPEC92 suite package. Version 2.0 of C compiler was used to compile all benchmark programs written in C. The MIPS pixie tool [25] was used to instrument the executable of a benchmark program. The resulting annotated executable file runs exactly the same as the original executable except that it also writes a stream of traces for both instruction and data to a special system file descriptor (i.e. 19), on which our cache simulators can capture each trace item and collect the statistics of interest.

For each C benchmark program, we ensure identical trace streams (i.e. workload) for both the baseline cache simulator and the *CAT* simulator. Once a benchmark program's executable and pixie-annotated executable are generated, we duplicate them together with all input files for the benchmark onto a pair of distinct DECStation 5000 workstations, where the simulator for the *CAT* cache and the simulator for the baseline cache run separately.

The first 2^{31} references (when available) were traced and simulated for all C benchmarks. This trace limit forces some programs such as Alvin, Ear and Xlisp not to run to completion. In the original SPEC92 suite evaluation procedure, some benchmark programs require multiple input data files to produce an average performance metric. In our simulation study, as long as multiple input files can be processed in a single pass of simulation run, we used all input files for the benchmark from the SPEC package. Such benchmarks include Alvin, Compress, Ear, Eqntott, and Xlisp. For other benchmark programs, we selected one representative input file and collected statistics for the benchmark after one pass of simulation run. For example, we picked file *cps.in* as input file for Espresso, *loada2* for Sc, and concatenation of all **.i* input files as the input file to Gcc(cc1). Table 1 lists some characteristics of the benchmark programs in our trace driven simulation studies.

4.2 Execution Driven Simulation

The execution driven simulator is developed based on IBM 370 mainframe processor which has a 16×128 special register file and multiple general purpose registers for high speed floating point

Programs	Alvin	Compress	Ear	Eqntott
Total trace	2, 147, 482, 980	112, 252, 893	2, 147, 482, 980	1, 560, 163, 944
Inst references (%)	78.96	80.98	81.64	86.13
Data references (%)	21.04	19.02	18.36	13.87
Programs	Espresso	Gcc(cc1)	Sc	Xlisp
Total trace	757, 371, 906	219, 779, 539	266, 262, 916	2, 147, 482, 980
Inst references (%)	82.58	1.66	0.18	0.03
Data references (%)	17.42	98.34	99.82	99.97

Table 1: Characteristics of C Programs in SPEC92

Programs	Doduc	Fpppp	Hydro2d	Mdljdp2	Mdljsp2
Data ref.	415,547,451	2,648,549,710	1,402,325,411	1,322,845,562	791,580,583
Programs	Nasa	Su2cor	Swm256	Tomcatv	Wave
Data ref.	1,767,622,115	1,242,211,078	1,082,503,422	666,046,023	635,640,782

Table 2: Trace sizes of Fortran programs in SPEC92

operations. The simulator emulates the instruction set of the IBM S/370 class processor, including memory accesses. Fortran benchmark programs are compiled using IBM 370 Fortran VS2.5 compiler with all optimization settings. The resulting assembly codes are fed into the execution driven simulator. During the execution process, whenever a memory reference is issued, the address of the data reference is passed to a cache simulator to collect performance statistics.

For all the Fortran benchmark programs, we use the standard input files provided within the original SPEC92 package. Because of the large register file that is available in the IBM 370, the trace sizes of the Fortran programs are considerably smaller. Therefore, we are able to simulate all Fortran programs for complete runs. Table 2 lists the trace sizes of the Fortran benchmark programs in our execution driven simulation studies.

5 Performance Results

We present numerical results from our simulation experiments in this section. Throughout our simulation experiments, performance of the *baseline cache* serves as a baseline reference for our comparative study of the *CAT* cache performance. Caches with data area capacities of 16K bytes and 64K bytes were simulated and 16-byte and 32-byte line sizes were used in various configurations. For the *CAT* cache, we consider *tag caches* with sizes ranging from 16 entries to 64 entries. We first evaluate the tag cache performance for LRU tag replacement policy and then discuss the least counter replacement.

Miss ratios are used as the performance metric in our following discussions. There are two types of misses: misses from the *tag cache* and misses from the data area of a cache. A *tag miss* is said to occur when the tag of a memory reference does not match any tag in the tag cache, which may result in tag replacement if the tag cache is full. A *data miss* is defined as usual, i.e. a referenced data is not in the data cache. *Tag miss* is used as a performance measure to evaluate how good the tag cache performs. Lower tag miss ratio indicates good address locality whereas high tag miss ratio may result in a lot of cache data invalidations due to tag replacements. As will be shown shortly, the tag miss ratios for our tag cache design are extremely small implying the high effectiveness of

Programs	16-tag CAT		32-tag CAT	
	tag % miss ratio	tag miss count	tag % miss ratio	tag miss count
Alvin	0.24	1, 091, 214	0.00	41
Compress	6.96	1, 486, 951	0.00	30
Ear	0.00	176	0.00	176
Eqntott	0.15	327, 975	0.06	134, 996
Espresso	0.00	1, 201	0.00	27
Gcc(cc1)	0.24	524, 469	0.09	193, 334
Sc	0.02	62, 671	0.00	33
Xlisp	0.00	10	0.00	10

Table 3: Tag misses in the tag-cache (16K-byte capacity and 16-byte line)

the CAT. The exact effect of such small tag miss ratio on data miss ratio is an important factor in evaluating the performance of CAT caches. To compare the data miss ratio of a *CAT* cache with the baseline cache, we measure *absolute difference* of data misses between the CAT cache and the baseline cache. Note that there is no tag cache in baseline cache. The absolute difference is defined as

$$\begin{aligned}\text{absolute difference} = \; & (\text{data miss ratio of CAT cache} - \\ & \text{data miss ratio of baseline cache})\end{aligned}$$

and *relative difference*, which is defined as

$$\text{relative difference} \;=\; \frac{\text{absolute difference}}{\text{data miss ratio of baseline cache}}$$

We start our experiment by simulating a 16K bytes *CAT* cache with the line size of 16 bytes. The tag cache size of the *CAT* cache is set to be 16 entries and 32 entries, respectively. We first examine the behavior of the tag cache itself. Table 3 lists the tag miss ratio and tag miss counts on the tag caches. As shown in Table 3, among the eight C programs, we observed that the tag miss ratios of 16-entry tag cache are all below 0.3% with an exception of Compress which has about 7% miss ratio. For the 32-entry tag cache, the tag miss ratios of all programs are close to zero. The tag misses (miss counts) are mainly cold start misses and misses during working-set transitions.

The data miss ratios of the same size baseline cache and *CAT* cache are shown in Tables 4 and 5. With 16-entry tag cache, except Compress, the data miss ratios of the *CAT* cache differ from those of the baseline cache by less than 0.8% (absolute difference). However, such small difference may convert to a noticeable relative difference if miss ratio of the baseline cache is very small as shown in the tables. For *CAT* caches with 32-entry tag cache, the miss ratio difference between the *CAT* cache and the baseline cache is hardly noticeable for all the 18 benchmark programs. For six benchmark programs, Alvin, Compress, Espresso, Sc, Xlisp, and Fpppp, the miss ratio of the *CAT* cache is exactly the same as that of baseline cache. The remaining programs have relative difference less than 1% except for Ear and Gcc. Note that our simulator did not run to completion for Ear due to the trace limit, which results in a miss ratio significantly smaller than that reported in [7]. The relative difference for several C programs can be attributed to the dynamic addressing structures. We observed in C programs that pointer-based indirect addressing are used by many algorithms handling dynamic data structures, such as routines manipulating text string or link lists. Through source code inspection, we found such dynamic addressing is indeed used extensively in both Gcc(cc1) and Eqntott. For the Fortran programs, on the other hand, we observed very high

Program	Baseline	CAT					
		16-tag			32-tag		
	%miss ratio	%miss ratio	%difference absolute	%difference relative	%miss ratio	%difference absolute	%difference relative
Alvin	8.04	8.6	0.56	6.96	8.04	0.0	0.0
Compress	15.78	19.82	4.04	25.6	15.78	0.0	0.0
Ear	0.32	0.39	0.07	21.88	0.38	0.06	18.75
Eqntott	7.18	7.59	0.41	5.71	7.2	0.02	0.28
Espresso	3.51	3.56	0.05	1.42	3.51	0.0	0.0
Gcc(cc1)	3.74	4.48	0.74	19.79	3.88	0.14	3.74
Sc	2.27	2.35	0.08	3.52	2.27	0.0	0.0
Xlisp	2.57	2.57	0.0	0.0	2.57	0.0	0.0

Table 4: Comparison of data miss ratios of C programs between the baseline cache and *CAT* cache (16K-byte capacity and 16-byte line size)

Programs	Baseline	CAT					
		16-tag			32-tag		
	% miss ratio	% miss ratio	% difference absolute	% difference relative	% miss ratio	% difference absolute	% difference relative
Doduc	4.73	4.79	0.06	1.27	4.77	0.04	0.85
Fpppp	0.52	0.53	0.01	1.92	0.52	0.00	0.00
Hydro2d	16.99	17.46	0.47	2.77	17.12	0.13	0.77
Mdljdp2	2.36	2.39	0.03	1.27	2.37	0.01	0.42
Mdljsp2	2.03	2.09	0.06	2.96	2.05	0.02	0.98
Nasa	19.30	19.73	0.43	2.23	19.48	0.18	0.93
Su2cor	22.44	22.67	0.23	1.02	22.58	0.14	0.62
Swm256	14.35	14.61	0.26	1.81	14.47	0.12	0.84
Tomcatv	9.13	9.23	0.10	1.10	9.22	0.09	1.00
Wave	6.82	6.89	0.07	1.03	6.85	0.03	0.44

Table 5: Comparison of data miss ratios of Fortran programs between the baseline cache and *CAT* cache (16K-byte capacity and 16-byte line)

spatial locality and fairly regular data access patterns. As a result, with 32-entry tag cache, the average absolute difference for all Fortran programs is close to zero and the relative difference is less than 1%, as shown in Table 5. When the tag cache size is reduced to 16 entries, the maximum absolute difference is 0.5% and maximum relative difference is 3%. The average absolute difference and relative difference are less than 0.2% and 1.7% respectively.

As indicated previously, replacements in the tag cache may cause invalidations of data lines in the data cache, which in turn may result in additional cache misses. In order to see the effects of such invalidations on cache performance, we have collected statistics on the number of cache misses caused by such invalidations. In Table 6, we listed cache misses caused by invalidations (column marked "inv. miss"), the total cache misses, as well as the ratio of the two. With 16 tags, we observed notable proportion of cache misses caused by invalidations, while with 32 tags there is no invalidation misses except for Eqntott for which we observed about 11.7% of the total misses.

Our results show that the *CAT* cache with 32-entry tag cache can capture the address locality in data references. The cache performance of the *CAT* cache with only 32 tags is approximately the same as the cache performance of the baseline cache having 1024 (for 16K byte cache) or 2048 (for 64K byte cache) tags.

As indicated in the introduction, one important trend in computer applications is the increased

Program	CAT					
	16-tag			32-tag		
	inv. miss	total miss	ratio	inv. miss	total miss	ratio
Alvin	3413061	38917710	0.088	0	36389713	0.0
Compress	3817269	4230646	0.90	0	3366339	0.0
Ear	0	1982673	0.0	0	1982673	0
Eqntott	7269418	15844576	0.46	1818847	15577806	0.117
Espresso	103974	4635843	0.022	0	4567342	0.0
Sc	348008	4752604	0.073	0	3724440	0.0
Xlisp	0	53113183	0.0	0	53113183	0.0

Table 6: Statistics of misses caused by invalidations due to tag replacements (16K-byte capacity and 16-byte line)

Programs	16-tag CAT		32-tag CAT	
	tag % miss ratio	tag miss count	tag % miss ratio	tag miss count
Alvin	0.00	9	0.00	9
Compress	0.00	8	0.00	8
Ear	0.00	46	0.00	46
Eqntott	0.00	144	0.00	41
Espresso	0.00	9	0.00	9
Gcc(cc1)	0.00	172	0.00	51
Sc	0.00	9	0.00	9
Xlisp	0.00	8	0.00	8

Table 7: Tag misses in the tag-cache (64K-byte capacity and 32-byte line)

demand for larger address space, the address word increases by more than one bit per year. The trend in computer technology is that multimillion transistors can be integrated on a single chip making it possible for computer designers to include larger on-chip cache memories for high memory performance. One immediate effect of such trends is an increased tag cost of on-chip caches since in the baseline cache organization both the number of tags and tag length will increase with these trends. In order to see whether our *CAT* cache design can last through the trends in computer applications and computer technology, we carried out additional simulations for larger on-chip caches.

Table 7 lists the tag cache performance for benchmarks on 64K-byte caches with 32-byte line size. Clearly, a 64K-byte *CAT* cache exhibits much better tag cache performance than 16K-byte *CAT* caches with the same number of tag entries. For example, benchmark Eqntott now has only 144 misses in the tag cache (refer to Table 6), a tremendous reduction from 327,975 misses on tag cache of the same size in a smaller *CAT* cache (refer to Table 3). This large amount of reduction in tag misses results mainly from tag shrinking. The tag length is reduced by 2 bits due to the cache size increase from 16K bytes to 64K bytes. One interesting observation from this result is that the information contents of high order bits of tags are significantly smaller than the information contents of the low order bits. This observation leads to the *Partial CAT* design whose performance will be shown shortly.

Our results also show that further increasing tag cache size from 32 to 64 does not bring any noticeable benefit. The 32-entry tag-cache has nearly identical performance as the large 64-entry

Programs	Baseline	CAT					
		32-tag			64-tag		
	% miss ratio	% miss ratio	% difference absolute	% difference relative	% miss ratio	% difference absolute	% difference relative
Alvin	3.36	3.36	0.00	0.00	3.36	0.00	0.00
Compress	11.90	11.90	0.00	0.00	11.90	0.00	0.00
Ear	0.22	0.22	0.00	0.00	0.22	0.00	0.00
Eqntott	3.67	3.67	0.00	0.00	3.67	0.00	0.00
Espresso	0.47	0.47	0.00	0.00	0.47	0.00	0.00
Gcc(cc1)	0.51	0.51	0.00	0.00	0.51	0.00	0.00
Sc	0.88	0.88	0.00	0.00	0.88	0.00	0.00
Xlisp	0.34	0.34	0.00	0.00	0.34	0.00	0.00

Table 8: Comparison of data miss ratios of C programs between the baseline cache and *CAT* cache (64K-byte capacity and 32-byte line)

Programs	Baseline	CAT					
		32-tag			64-tag		
	% miss ratio	% miss ratio	% difference absolute	% difference relative	% miss ratio	% difference absolute	% difference relative
Doduc	1.02	1.03	0.01	0.98	1.02	0.00	0.00
Fpppp	0.07	0.07	0.00	0.00	0.07	0.00	0.00
Hydro2d	6.86	6.92	0.06	0.87	6.89	0.03	0.44
Mdljdp2	0.91	0.93	0.02	2.20	0.91	0.00	0.00
Mdljsp2	0.61	0.62	0.01	1.64	0.61	0.00	0.00
Nasa	12.13	12.26	0.13	1.07	12.19	0.06	0.49
Su2cor	6.50	6.55	0.05	0.77	6.53	0.03	0.46
Swm256	6.15	6.20	0.05	0.81	6.17	0.02	0.33
Tomcatv	5.62	5.65	0.03	0.53	5.63	0.01	0.18
Wave	1.57	1.58	0.01	0.64	1.57	0.00	0.00

Table 9: Comparison of data miss ratios of Fortran programs between the baseline cache and *CAT* cache (64K-byte capacity and 32-byte line)

tag-cache. The data miss ratios of the 64K-byte cache are shown in Tables 8 and 9. For all the C benchmarks, the *CAT* cache shows the exactly the same cache performance as that of the baseline cache. It is further shown that 32-entry tag cache is sufficient to hold all distinct tags of cached data since increasing tag size to 64 entries results in the same performance. For Fortran programs, we can also draw a similar conclusion. The performance difference between the *CAT cache* and the baseline cache organizations is hardly noticeable.

Instruction Cache: We have also collected traces for instruction caches in our experiments. The address locality of instruction references is even better than data references. With 16-entry or 32-entry tag cache, the instruction *CAT* cache shows identical cache performance as the baseline cache of the same cache size. In other words, there is NO difference in terms of miss ratios between a 16-tag instruction *CAT* cache and 1024-tag or 2048-tag baseline instruction cache.

LCR ***CAT*****:** As indicated in Section 3, in addition to the spatial and temporal locality of tags, there is one more factor that may affect *CAT cache* performance. This factor is the relative significance of cached tags. Such relative significance is reflected by the number of cached data lines that are associated with a tag. One can reasonably believe that better cache performance can be obtained by always replacing the tag having the least number of corresponding cached data lines when replacement is necessary. This consideration leads to the *least counter replacement* (LCR)

Programs	16-tag CAT	
	% miss ratio with LRU	% miss ratio with LCR
Doduc	4.79	4.78
Hydro2d	17.46	17.52
Mdljdp2	2.39	2.37
Mdljsp2	2.09	2.05
Nasa	19.73	24.21
Su2cor	22.67	23.18
Swm256	14.61	14.82
Tomcatv	9.23	9.22

Table 10: Comparison of data miss ratios of Fortran programs between the 16-tag *CAT* with LRU and that with LCR (16K-byte capacity and 16-byte line)

Programs	16-tag CAT	
	% miss ratio with LRU	% miss ratio with LCR
alvin	8.6	8.22
compress	19.82	25.80
ear	0.39	0.37
eqntott	7.59	19.74
espresso	3.56	3.92
sc	2.35	2.63
xlisp	2.57	2.57

Table 11: Comparison of data miss ratios of C programs between the 16-tag *CAT* with LRU and that with LCR (16K-byte capacity and 16-byte line)

policy as introduced in Section 3. However, the real situation is not so straightforward. We carried out simulation for the *CAT* cache with the least counter replacement policy. In our simulation, we assume LCR policy using *most recently touched* bit. That is, we apply LCR policy to tag cache entries which are not shared by the most recently accessed cache line. Tables 10 and 11 show the performance of the *CAT* cache with this new tag replacement policy on some Fortran and C programs in SPEC92 suite. As Table 11 indicates, for SPEC C programs, we observe minute difference in performance between two policies. For FORTRAN programs, we found that LCR policy does result in slight performance improvement for 4 floating point programs, namely Doduc, Mdljdp2, Mdljsp2 and Tomcatv, as shown in Table 10. All these programs have code fragments with intensive floating point operations on large data arrays within tight loops. For such access patterns, LCR policy is able to retain the tags for array variables in the tag cache, therefore reducing performance degradation which would otherwise be incurred due to invalidations of a large number of cache lines upon tag replacement. For other SPEC92 FORTRAN programs, LCR policy does not bring about improvement over LRU policy. We notice that some programs even show worse performance than using the simple LRU policy, such as Nasa, Su2cor, Swm256 and Hydro2d.

From code inspection, we notice that these programs mostly involve operations on scalar variables or manipulations on volatile dynamic data structures. For these programs, the reference counter becomes less significant. Furthermore, one most-recently-touched bit in the LCR algorithm is not sufficient to record the reference history, particularly for a 16-way associative tag cache. As a result, there are cases where tags for the most frequently used data are replaced from the tag cache due to their smaller counter values. Therefore, the LRU *CAT* shows slightly better performance than the LCR *CAT* for these programs.

Programs	Baseline	8-tag ParCAT		
	% miss	% miss	% difference	
	ratio	ratio	absolute	relative
Doduc	4.73	4.74	0.01	0.21
Fpppp	0.52	0.52	0.00	0.00
Hydro2d	16.99	17.02	0.04	0.24
Mdljdp2	2.36	2.36	0.00	0.00
Mdljsp2	2.03	2.04	0.01	0.49
Nasa	19.30	19.32	0.02	0.10
Su2cor	22.44	22.48	0.04	0.18
Swm256	14.35	14.37	0.02	0.14
Tomcatv	9.13	9.16	0.03	0.33
Wave	6.82	6.82	0.00	0.00

Table 12: Comparison of data miss ratios of Fortran programs between the baseline cache and ParCAT cache (16K-byte capacity and 16-byte line size)

Partial *CAT*

We observed through our experiments that a small part of address tag changes more frequently during program executions while the remaining part keeps almost unchanged. In order to reduce the tag cache size and exploit address locality further, we partition tags into two parts. We store the low order part of the tags in its original form, i.e. there is a one-to-one correspondence between the low order part of the tags and the cache lines. Only the remaining high order tags are cached in the tag cache. We call this organization *partial CAT* cache (ParCAT). Table 11 tabulates the performance results of partial *CAT* cache with cache size being 16K bytes and line size 16 bytes. In ParCAT, a small tag portion (8-bit) is attached to each cache line. The tag-cache size is reduced to 8-entry. We observed that the ParCAT caches with only 8 tag entries perform better than regular *CAT* caches having the same cache capacity but with 16 tag entries and 32 tag entries. The actual area savings of ParCAT may not be much more than the regular *CAT* cache because the low order bits need to be stored with each of 1K cache lines. The partial *CAT* does, however, provide an opportunity for computer designers to further exploit the design tradeoffs.

Summary of Performance Analysis

We observed the evident address locality of tags in consecutive references for both instruction and data. For the SPEC92 benchmark suite, it is observed that the *CAT* cache design using 32-entry tag cache can provide the best performance affinity to the baseline cache, for both instruction and data caches. While achieving comparable cache performances, the *CAT* cache reduces the number of on-chip tags by 32 or 64 times. The on-chip area made available by our *CAT* design can be utilized for more critical functions including large cache capacity.

6 Tag Cost Analysis

An important cost measure for on-chip cache is its occupied silicon area (or chip-area). The reduction in terms of number of tags in *CAT* cache may not directly reflect the actual savings in terms of chip-area. In this section, we carry out a simple cost analysis for two tag memory implementations in the *CAT cache* and the baseline cache, respectively, using the area model for on-chip memories

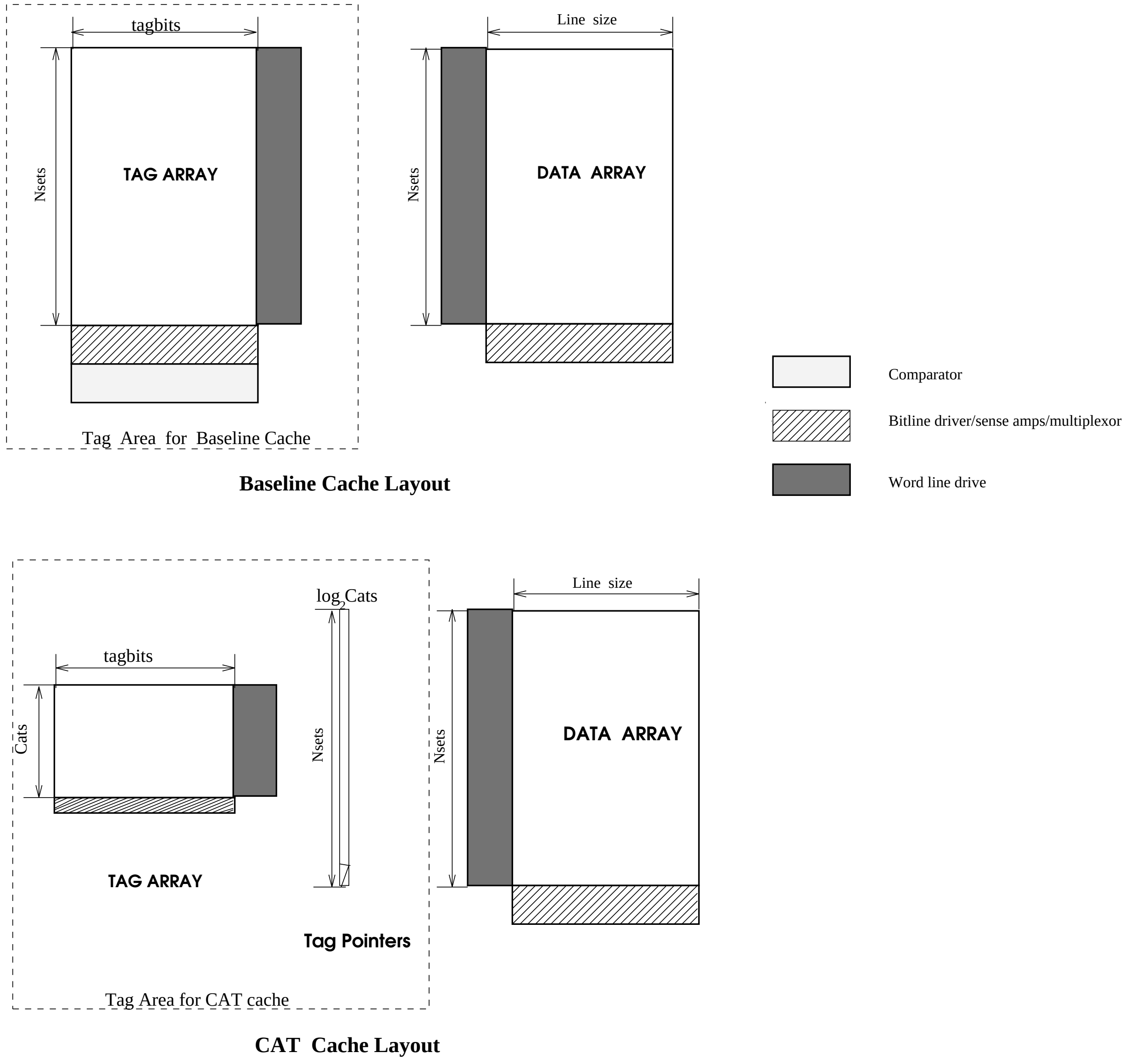

Figure 7: Simplified Cache Area Layout for Baseline Cache and CAT Cache

presented by Mulder, Quach and Flynn (referred to as MQF model) [4]. The chip-area required by each part of a cache organization includes memory cells, drivers, sense amplifiers, address tags, dirty and valid bits, comparators and control logic. Taking into account the effects of bandwidth on different memory cell types (i.e. register cell, SRAM cell and DRAM cell), the MQF model gives a good approximation of the total cost for a given memory structure. Mulder et al. compared MQF model's area prediction with twelve actual processor designs and found typical errors under 10% and a maximum error of 20.1%. A technical-independent notation of a *register-bit equivalent (*rbe*)* is defined as a unit of chip-area. The area unit, *rbe*, equals the area of a six-transistor static register cell. It was assumed in the model that a sense amplifier on a bit line, control line driver on a word line and a bit of tag comparator each take a chip-area approximately equal to 6 rbe. For cache memory, each cell (SRAM) is equivalent to 0.6 rbe.

Based on the internal structure of an SRAM cache organization, a simple cache area layout for both baseline cache and *CAT* cache is shown in Figure 7. Data array area keeps unchanged in CAT cache compared to the baseline cache with identical cache parameters. Therefore our cost analysis is performed only on tag area. The CAT cache and the baseline cache in this section are all direct-mapped caches. Similar to the notations used in the MQF's model [4], let $tagbits$ be the length of a tag in bits and $Nsets$ be the number of sets in the cache. For the baseline cache, $Nsets$ is also the total number of tags. Ignoring the PLA cost, the tag area cost in terms of rbe for the baseline cache is given by [4]

$$tag - area_{old} = 0.6(tagbits + 6)(Nsets + 6 + 6) \text{ rbe} \tag{1}$$

In the CAT cache, the tag array consists of *tag cache* which is a fully associative cache holding distinct address tags, and *tag pointers*. The number of tags in CAT cache is no longer the same as the number of cache lines. The number of tags in the *CAT cache* is the size of the *tag cache* denoted by Cat_s. In addition to the area for the tag cache, extra on-chip area needs to be allocated to *tag pointers*. The tag pointers work as a RAM memory for all read and write operations. In an event of tag replacement, all the valid bits of the cache lines associated with the victim tag should be cleared, which makes the tag pointers fully associative. The nine transistor CAM cell [24] consisting of storage and a comparison circuit can be used to implement the associative memory. Note that the number of criss-crossing wires besides the number of transistors can also affect chip area estimation. Similar to Alpert's assumption [26], the size of each memory cell in the tag area is assumed to be twice the size of a SRAM cell in the cache. The width and the length of each cell are equal to $\sqrt{2}$ if the CAM is implemented as a square cell. Assuming that *tag pointers* are driven by the same set of drivers (index decoders) as data cache, the tag area of CAT cache is

$$\begin{aligned} tag - area_{new} &= tag\ pointers\ area + tag\ cache\ area \\ tag - area_{new} &= 0.6(\sqrt{2}Nsets + 6)(\sqrt{2}log_2 Cat_s) + \\ &\quad 0.6(\sqrt{2}tagbits + 6)(\sqrt{2}Cat_s + 6) \text{ rbe} \end{aligned} \tag{2}$$

where Cat_s is the size of the tag cache in terms of number of tags cached. If we take the ratio of Equation (1) to Equation (2), we have

$$\frac{tag - area_{old}}{tag - area_{new}} = \frac{0.6(tagbits + 6)(Nsets + 12)}{1.2(Nsets + 4.2)(log_2 Cat_s) + 1.2(tagbits + 4.2)(Cat_s + 4.2)} \tag{3}$$

The area savings as a percentage of total cache area are shown in Figure 8. It can be seen from this figure, as the address word length increases the area saving increases.

For very large cache or a large $Nsets$, we have

$$\frac{tag - area_{old}}{tag - area_{new}} \approx \frac{(tagbits + 6)}{2log_2 Cat_s}, \quad when\ Nsets \to \infty. \tag{4}$$

For example, if we had 32-entry tag cache on a 64-bit machine that has 16K-byte on-chip cache, the area saving would be about 5.6 times.

If the address space is much larger, i.e. very large $tagbits$, we have

$$\frac{tag - area_{old}}{tag - area_{new}} \approx \frac{(Nsets + 12)}{2(Cat_s + 4.2)}, \quad when\ tagbits \to \infty. \tag{5}$$

For the same machine above with line size of 16 bytes, the area savings in implementing the tag array would be an order of magnitude.

In real systems, the CAT cache may have additional area cost reductions because of the following reasons. One major factor that contributes to the cost of a CAM is the requirement of each

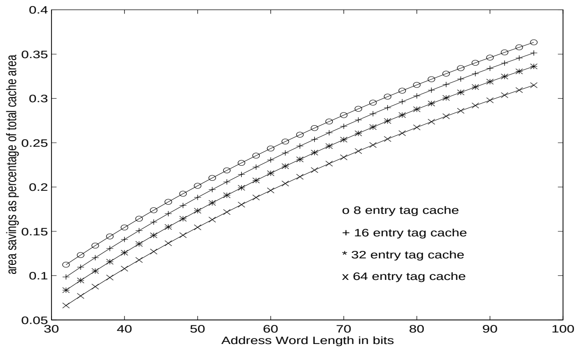

Figure 8: Area savings as a percentage of total cache area (cache size: 16K byte, line size: 16 byte) (CAM cell: height=width)

individual bit to drive wires of length proportional to word length in order to generate a matched address. In order to minimize the time needed for this address generation, driver trees that consist of hierarchy of transistors are generally implemented [27]. The word size in our *tag pointer* memory is anticipated to be very small. Each pointer word usually consists of only a few bits between 3 and 6. A match signal is used to clear the valid bit rather than driving encoder and multiplexer circuits that are necessary for associative search in set-associative cache [23]. Furthermore, matching operations in the *tag pointer* memory are performed only on a cache miss, which is not a part of critical path length. Similarly, associative search in the *tag cache* is necessary only when a miss occurs. Therefore, the expensive and high speed driver tree is not necessary. Under this situation, the CAM cell size in *CAT* cache should be smaller than twice the size of SRAM cell in *baseline cache* as we assumed previously. Furthermore, we note that in real systems the pitches of the CAM and SRAM cells are the same. Rather than having the square CAM cell with a $\sqrt{2}$ height, we might want the CAM cell to have the same height as the SRAM cell but doubling the width. This is because the square CAM cell with a larger pitch than SRAM incurrs the wasted space between the cache lines for the data array. In order to minimize the overall chip-area cost including both tag area and data array, the designer might prefer using the second geometry parameters (i.e. height=1, width=2) for CAM cells.

7 Conclusions and Future Work

In this paper, we have investigated the problem of how to effectively reduce on-chip tag overhead without adversely affecting cache performance. The address locality property among memory references was studied through extensive simulation experiments on SPEC92 benchmark suite. As our experiments demonstrate, the tag field in memory addresses exhibits extraordinary address locality. A large amount of cache lines in a cache shares just a small number of distinct address tags. Based on this observation, we proposed a novel cache design called *CAT cache* (*caching address tags*). The *CAT cache* exploits address locality property in order to reduce on-chip tag area via eliminating the tag redundancy. Our studies show that a *CAT cache* with only 16-entry or 32-entry tag cache can deliver performance identical to an ordinary cache organization with 1024 or 2048 tags in terms of cache miss ratios. Using an established area model for on-chip memory designs, we have analyzed the on-chip area savings resulting from *CAT cache* design and found that the savings are significant. Such savings will increase as the address space of a processor grows.

Researchers [5] have suggested that it is important to fine-tune the chip-area allocation strategies driven by performance considerations. In order to effectively support new application environments, it is shown in [5] that increasing TLB size and I-cache capacity is preferred to increasing D-cache capacity in single-chip processor designs. Our *CAT* cache design offers an alternative approach for reducing chip-area cost of the D-cache without affecting cache capacity. The area savings resulting from eliminating tag area redundancy introduce an extra dimension to the tradeoff evaluation equations in performance-driven area allocation strategies.

The work presented in this paper can be extended in several ways. Both designs for set-associative cache presented in Section 3 have advantages and disadvantages. Additional study is necessary to come up with an optimal *CAT* in terms of performance and cost. It is also possible to devise a more efficient replacement scheme for the *tag cache* by taking into account the distinct features of the *tag cache* from regular caches. Finally, we are currently interested in different ways of linking the tags in the *tag cache* with the data lines. Since the *tag cache* is an order of magnitude smaller than the data area, further cost reduction is possible if pointers can be stored in the *tag cache* in a subcube form instead storing them with cached data.

References

[1] R. L. Sites, "Alpha AXP architecture," *Digital Technical Journal*, vol. 4-4, pp. 19–34, 1992.

[2] F. Okamoto and et al., "A 200-MFLOPS 100-MHz 64-b BiCMOS vector-pipelined processor (VPP) VLSI," *Journal of Solid State Circuits*, vol. 26, pp. 1885–1892, Dec. 1991.

[3] *PowerPC 601, RISC Microprocessor User's Manual.* Motorola, 1993.

[4] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *Journal of Solid State Circuits*, vol. 26, pp. 98–106, Feb. 1991.

[5] D. Nagle, R. Uhlig, T. M. Mudge, and S. Sechrest, "Optimal allocation of on-chip memory for multiple-API operating systems," *The 21st Ann. Int. Symp. on Comp. Arch.*, pp. 358–369, April 1994.

[6] M. Farrens, G. Tyson, and A. R. Pleszkun, "A study of single-chip processor/cache organizations for large number of transistors," *The 21st Ann. Int. Symp. on Comp. Arch.*, pp. 338–347, April 1994.

[7] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache performance of the SPEC92 benchmark suite," *IEEE Micro*, pp. 17–27, Aug. 1993.

[8] Q. Yang and S. Adina, "A one's complement cache," *Proceedings of 94' Int'l Conf. on Parallel Processing*, pp. 250–258, Aug. 1994.

[9] Q. Yang, "Introducing a new cache design into vector computers," *IEEE Trans. on Computers*, vol. 43, Jan. 1994.

[10] J. L. Hennessy and N. P. Jouppi, "Computer technology and architecture: An evolving interaction," *IEEE Computer*, pp. 18–29, Sept. 1991.

[11] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *10th Annu. Symp. on Comput. Arch.*, pp. 124–132, 1983.

[12] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th. Int'l Symp. on Comp. Arch.*, pp. 54–63, 1991.

[13] J. Torrelas, M. S. Lam, and J. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Trans. on Computers*, vol. 43, pp. 651–663, June 1994.

[14] S. J. Eggers and R. H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," in *Proc. 3rd ASPLOS*, pp. 257–270, April 1989.

[15] C. Dubnicki and T. LeBlanc, "Adjustable block size coherence caches," *19th Ann. Int'l Symp. on Computer Architectures*, May 1992. Queensland, Australia.

[16] Q. Yang, L. Bhuyan, and B.-C. Liu, "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor," *IEEE Trans. on Comput..*, vol. 38, pp. 1143–1153, August 1989. Special Issue on Distributed Computer Systems.

[17] *TMS390Z55 Cache Controller, Data Sheet.* Texas Instrument, 1992.

[18] A. Seznec, "Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio," in *The 21st Ann. Int. Symp. on Comp. Arch.*, pp. 384–393, April 1994.

[19] D. Hammerstrom and E. Davidson, "Information content of CPU memory referencing behavior," in *Proc. of 4th Annual Symposium on Computer Architecture*, pp. 184–192, March 1977.

[20] A. R. Pleszkun, B. R. Rau, and E. S. Davidson, "An address prediction mechanism for reducing processor-memory address bandwidth," in *Proc. of 1981 IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 141–148, November 1981.

[21] M. Farrens and A. Park, "Dynamic base register caching: a technique for reducing address bus width," *The 18th Ann. Int. Symp. on Comp. Arch.*, pp. 128–137, May 1991.

[22] N. P. Jouppi, "Cache write policies and performance," in *20th Ann. Int'l Symp. on Computer Architectures*, pp. 191–201, May 1993.

[23] M. D. Hill, "A case for direct-mapped caches," *IEEE Computer*, pp. 25–40, Dec. 1988.

[24] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Addison-Wesley, 1985.

[25] M. Smith, "Tracing with pixie," *Technical Report CSL-TR-91-497*, Nov. 1991.

[26] D. Alpert, "Memory hierarchies for directly executed language microprocessors," in *Tech. Report 84-260*, 1984. Computer Systems Lab., Standford Univ.

[27] C. Mead and L. Conway, *Introduction to VLSI systems.* Addison-Wesley, 1980.