**Dual Data Cache (Report): -**

**Abstract**

The Scalar Processors have a performance setback when it comes to Vector(Array) processing and other vector applications because of the Scalar processors limitation of processing single instruction at a time. The two most important reasons holding them back are: the use of same organization caching both spatial and temporal locality and the generally implemented "eager" caching policy. This leads to the well-known trade-off of designing caches with more number lines or implementing design which has large line size.

The remedy to such trade-off is a data cache organisation system called **Dual Data Cache**. Implementation of such cache design leads to efficient exploitation of both spatial and temporal localities in an efficient manner. The Dual Data Caching technique implement lazy caching technique which is a caching strategy that loads data into the cache when it is mandatory and that is achieved by the use of *locality prediction table* which maintains a history of details regarding the most recently executed load/store instruction.

Brief: -

• Text and data are not accessed randomly

• **Temporal locality**

– Recently accessed items will be accessed in the near future (e.g.,

code in loops, top of stack)

• **Spatial locality**

– Items at addresses close to the addresses of recently accessed items

will be accessed in the near future (sequential code, elements of

arrays

**Basic Functionality: -**

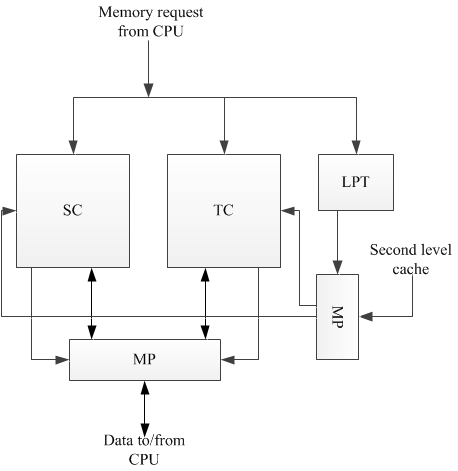
The Dual scheme is the first study that exploits temporal and spatial localities in two separate organizations.

1. Two independent caches coexist in the L1 cache: temporal and spatial subcaches. The line size in the spatial cache (several words) is greater than in the temporal cache (just one word). So, the temporal cache can only exploit the temporal locality whereas the spatial cache can exploit both kinds of localities. **Both subcaches share a common L2 cache.**

2. The scheme processes line classification at runtime using a **large hardware history table** to predict the expected data-locality type. The prediction can be spatial (the processor caches the data item in the spatial subcache), temporal (the processor caches the data in the temporal subcache), or bypass (the processor does not cache the data item).

**Simulation results show better performance compared to conventional cache systems**

**Block Diagram of Dual Data Cache: -**



(CPU – central processing unit; SC – spatial sub-cache; TC - temporal sub-cache; LPT – locality prediction table.)

**Among the different types of locality, the two main categories are spatial and temporal. Some memory references exhibit both, whereas others have only one of them, and some have none. The simplest way to exploit spatial locality is by means of very large cache lines, whereas the most effective way to exploit references with only temporal locality is by means of one-word lines. For memory references without locality, the most effective way to handle them is to bring only the required data (as opposed to a whole cache line) and not store it in the cache, to avoid polluting it. All these were the main goals of the Dual Data Cache.**

**LPT (Locality Prediction Table)**

A key part of the Dual Data Cache is the scheme that decides in which subcache each reference is placed. For this purpose, we proposed a structure called the Locality Prediction Table (LPT), which **keeps track at runtime** of the behaviour of memory instructions and tries to estimate their locality properties. The LPT is based on capturing the stride exhibited by memory instructions. Instructions with strides smaller than the Spatial Cache line size are mapped into the Spatial Cache. Instructions with zero stride (e.g., scalar references) and instructions with strides larger than the Spatial Cache line size that do not interfere with themselves when placed in the Temporal Cache are stored in the Temporal Cache. Other references simply bypass the two subcaches because they have poor locality or they would interfere when placed in cache. Besides, the Spatial Cache uses a very simple next-block prefetching scheme, which is very effective for this type of references.

**How LPT is used: -**

•If the required data is only in one of the subcaches, the data is read or written in that

subcache. This is a cache hit

• If the required data is found in both subcaches, it is read from any of them or written into

both two. This is again a cache hit.

• If the required data is not in any subcache, a cache miss occurs. In this case, the processor

is stalled and the required data is brought from the second-level cache (or main memory).

This data may be placed on just one of the two subcaches or may be not cached anywhere,

depending on predicted type of locality for this memory access. This prediction is made

by means of a small history table, which is called locality prediction table, whose

operation is explained later on. If spatial locality is predicted for this memory reference,

then the missed data is placed on the spatial cache. If temporal locality is predicted and

the referenced element is an scalar or if it belongs to a vector that can be placed on the

temporal cache without self-interference, the data is placed on the temporal cache.

Otherwise, it is not cached anywhere. Even if a vector exhibits temporal locality (but no

spatial locality), caching it would be detrimental if it interfere with itself.

The predicted locality for a memory reference is based on guessing whether the accessed data is an scalar, or an element of a vector, and in the case of being a vector element, it also depends on the stride and the size of the vector. These attributes are estimated by means of the locality prediction table. The locality prediction table is based on the history table that was proposed by Baer and Chen], Fu, Patel and Janssens, and Jegou and Temam as part of a hardware mechanism for prefetching vector data. The locality prediction table has a small number of entries. Each entry contains information about a recently executed load/store instruction. This information consists of the following fields:

a) Instruction address: The address of the load/store instruction. This field is the identifier

of the entry.

b) Last address: The last data address referenced by that instruction.

c) Stride: The difference between the last address and the second last address referenced.

d) Length: The number of consecutive elements accessed with by this load/store instruction

with the same stride.

e) Prediction: The locality prediction for this load/store instruction. The prediction can be:

spatial (i.e., in case of miss, cache it in the spatial cache), temporal (i.e., in case of miss,

cache it in the temporal cache) or don’t-cache (i.e., in case of miss, do not cache it

anywhere)

Every time there is a cache miss, the locality prediction table is used in order to decide how to cache the missing data. The locality prediction table is accessed at the same time as the cache memory. If the load/store instruction that caused the cache miss is not in the locality prediction table, the missing reference will be stored in the temporal cache once it is brought from the second-level cache (or main memory). Otherwise, when the load/store instruction is found in the locality prediction table, the information in the prediction field determines where the missing data is placed.

***Figure 2* shows the required hardware to implement the locality prediction table. Thehardware consists mainly of a table, a substracter, three full comparators, an equality comparator, a shifter and several multiplexers.**

The locality prediction table is managed as a cache. It has a small number of entries ant it

may be *direct mapped, set associative or fully associative*. Every time a load/store instruction is

executed the locality prediction table is looked up. If the instruction address is not in the table,

one of the entries of the table is allocated to this instruction and it is initialized as follows:

a) the address of the instruction,

b) the address of the referenced data,

c) stride equal to zero,

d) length equal to zero, and

e) prediction equal to temporal.

The instruction address and initial address field will maintain the same value as long as the entry remains in the table. The other field may vary every time the same load/store instruction is executed, as it is explained below. The last address and stride fields are always updated with the last referenced address and the difference between the last referenced address and the current one respectively. The length is incremented by one if the current stride is equal to the last one or it is reset to 1 if the stride is different.

If the current stride is equal to the last stride, the prediction field maintains the same value. Otherwise, the prediction field is updated as follows. If the last stride was equal to zero and the current stride is “small” (relatively to the line size of the spatial cache; this means to compare it with a constant that is implementation dependent; a suggested value could be half of the line size of the spatial cache), the prediction becomes spatial. If the last stride was zero and the current stride is “large” (larger than an implementation dependent bound) the prediction becomes don’t-cache. If the last stride was not zero and the current stride is different from the last stride, then the prediction depends on the vector length, the last stride, and the family to which the last stride belongs. The stride family defined by x is the set of strides σ•2 x with σ odd [6, 11]. If the last stride is “small” the prediction does not vary. Otherwise, If the vector length multiplied by 2 x is not higher than the size of the temporal cache, the prediction is temporal; otherwise the prediction is don’t cache, since the vector will interfere with itself.

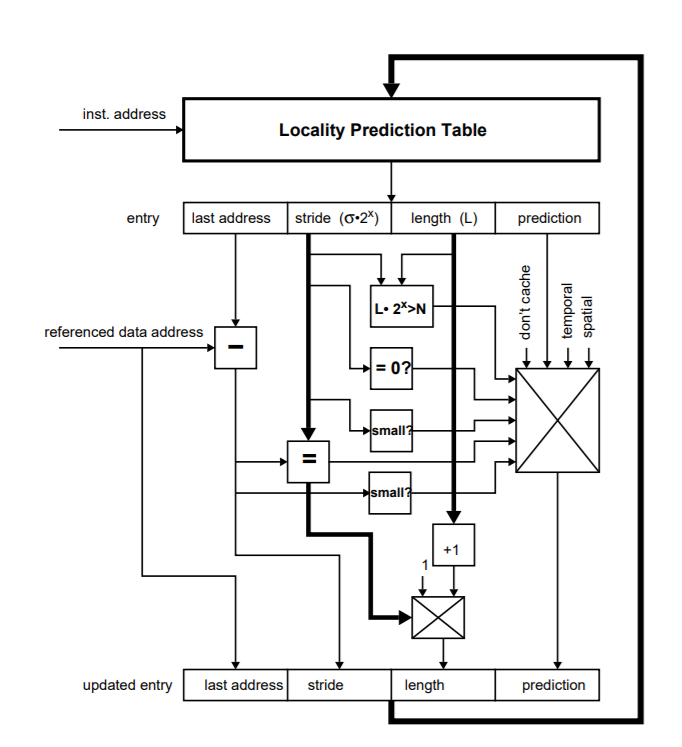


Figure 2

**PERFORMANCE EVALUATION**

The performance of the dual cache has been evaluated by simulating the execution of the

following set of benchmarks

• Comp: The Unix compress utility. It is one of the SPEC CINT92 benchmarks

• If: A do-loop with conditional statements that alternates different strides for accessing a

vector. It has been obtained from [7].

• Mm: A matrix by matrix multiplication.

• Bmm: This is a blocked version of a matrix by matrix multiply that in addition makes use

of copying.

• Spmv: Sparse matrix by vector

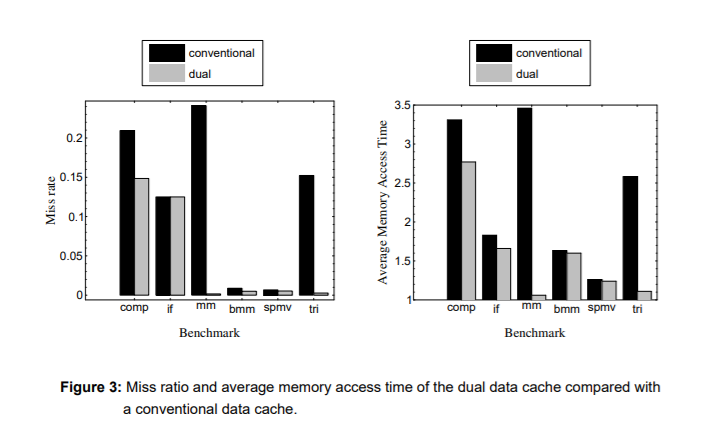
• Tri: A triangular matrix by matrix multiplication.

These benchmarks were compiled and executed on a DEC System 5810 with a MIPS R3000

processor. An exhaustive trace of their execution was been obtained with the pixie utility [9]. This

trace was fed into both a simulator of a conventional data cache memory and a simulator of a dual

cache



**CONCLUSIONS**

A novel data cache design for (super)scalar processors has been presented. The proposed data cache, called dual data cache, has two independent parts: the spatial cache, which has been designed to exploit both spatial and temporal locality, with the emphasis in the former, and the temporal cache which exploits only temporal locality. In other words, the proposed scheme is away of having large lines, which are convenient to exploit spatial locality, and small lines, which are more suitable if spatial locality is not present, in the same cache memory. The performance figures obtained for a set of benchmarks show that the dual data cache out performs a conventional cache. In some cases, like matrix multiplication, the improvement is spectacular, since the reduction in the average memory access time is about 70%. The main benefits of the dual cache are obtained for vector codes, for which a conventional cache usually fails to deliver good performance. However, even for non-vector codes, the dual data cache improves the performance of a conventional cache. In the case of the compress benchmark the improvement in the average memory access time was about 20%.We have also shown that for those cases where software techniques can be applied to improve the locality of a given algorithm (blocking and copying), the performance of the dualdata cache is not degraded. For instance, in the case of matrix multiply with blocking and copying, the average access time of the dual cache was still 2% lower than that of a conventional cache.

**References to Research Paper: -**

* <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=865890>
* <https://patents.google.com/patent/US5898852A/en>
* <https://www.sciencedirect.com/science/article/pii/S0065245814000060>
* <https://pdfs.semanticscholar.org/a53f/e9826e508fb4b0cd200ae06b102644ec9c41.pdf>