Application Specific Data Trace Cache Design

Sunita Parashar
Haryana College of Tech. & Mgmt,
Kaithal, Haryana

Anshu Parashar
Haryana College of Tech. & Mgmt,
Kaithal , Haryana

ABSTRACT
Increasing disparity between processor speeds and memory access times is a major problem in today's systems. In this paper we have studied the design of a proposed system i.e. An Application specific Data Trace Cache and have tried to do some modification to the Data Remapping portion of it. We have tried to further improve the benefits of this system by implementing this system with the help of an already existing concept of Cache Conscious Data Structures. Reference locality of data can be improved by changing a program's data organization and layout. This is the concept behind both cache conscious data structures as well as Data Trace Cache. So we have tried to merge both these strategies to get some more performance benefits.

General Terms
Performance, Design

Keywords
Trace Cache, Cache Conscious data structure, reference locality

1. INTRODUCTION
The memory hierarchy has been the central component in the design of computing platforms .It has widely served to bridge the performance gap between processor and supporting memory subsystem, usually by employing deep cache hierarchies where each level trades off capacity for access speed [1]. As processors are increasingly used in the context of embedded systems, the cost of this memory hierarchy is increasingly becoming a limiting factor in its ability to play as central a role.

Trace cache is a mechanism to enable low latency, high bandwidth instruction fetching. Trace caches store programs in a representation that is a hybrid of the static program representation and the dynamic instruction stream. Traces are snapshots of short segments of the dynamic instruction stream that are cached [2]. When a dynamic path is taken repetitively, instructions are provided from the trace cache, yielding a contiguous block of dynamic instructions that may correspond to noncontiguous blocks of code from the static representation.

2. RELATED WORK
As the rate of instruction execution increases, performance of the memory hierarchy becomes the bottleneck for most applications. In the past, the principal challenge in memory hierarchy management has been overcoming latency, but blocking and pre-fetching have improved that problem significantly [3]. As exposed memory latency is reduced, bandwidth has become the dominant performance constraint because limited memory bandwidth bounds the rate of data transfer between memory and CPU regardless of the speed of processors or the latency of memory access. So, program performance is now limited by effective bandwidth, that is, the rate at which operands of a computation are transferred between CPU and memory.

Currently, the principal software mechanism for improving effective bandwidth in a program, as well as reducing overall memory latency, is increasing temporal and spatial reuse through program transformation. Temporal reuse occurs when multiple accesses to the same data structure use a buffered copy in cache or registers, eliminating the need for repeated accesses to main memory. While temporal reuse reduces the frequency of memory accesses, spatial reuse improves the efficiency of each memory access by grouping accesses on the same cache line. Since most current machines transfer one cache line at a time from memory, this grouping amortizes the cost of the bandwidth over more references. The combination of temporal and spatial reuse can minimize the number of transferred cache lines, i.e. the total memory bandwidth requirement of the program. A substantive portion of the research on compiler memory management has focused on increasing temporal and spatial reuse in regular applications. Cache and register blocking techniques group computations on data tiles to enhance temporal reuse [4, 5]. Various loop reordering schemes seek to arrange stride-one data access to maximize spatial reuse [6, 7, 8]. Data transformations can often be used to effect spatial reuse when computation transformation is insufficient or illegal [9]. Data remapping can be used to improve cache performance by observing MAPs, and packing in contiguous memory locations sequentially accessed patterns to improve spatial locality. The above techniques adopt data layouts which are cache conscious [10]. Cache performance can also be improved by changing the organization and layout of its data—even complex, pointer-based data structures. Techniques like *structure splitting* and *field reordering*— improve the cache behavior of structures larger than a cache block.[11].

2.1 Approach 1: Data trace cache
Modern Superscalar processors with high Instruction Per Cycle rates require enough supply of instructions for execution in parallel and that in turn requires concurrent access to different cache blocks. For this a multiport cache can be used but this is an expensive solution both in terms of power as well as area. To solve the problem, Instruction Trace Cache has been designed which pack instructions from different cache blocks in to a single cache line in their dynamic execution order.

As enough instructions are required in superscalar processors, same is the case with the data. But multiported Data cache will be prohibitive in terms of cost. So, we need some way to have a virtual multiport data cache. This type of data cache is designed with the concept of instruction trace cache. This has been called as Data Trace Cache and like Instruction Trace Cache it packs together data from diverse addresses that are required for concurrent access during program execution.

The new system has been designed with a Data Trace Cache, a level-1 data cache and a cache controller. This new memory system is shown in fig 1. The cache controller has four read ports and a write port .If CPU required only one read port at a given cycle the cache controller request the data directly from level-1 data cache, which has only single

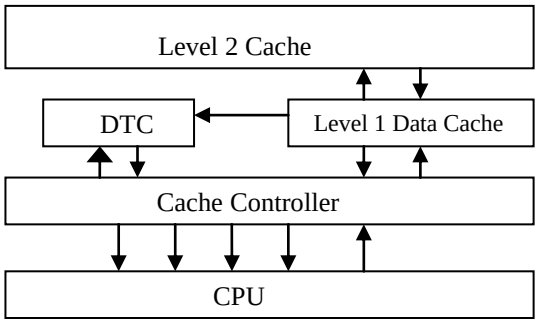

Fig 1. Memory System

read port. In case the CPU require data from more than one address the cache controller request all data from entire set of addresses from DTC. Although the DTC has only one read port, each DTC port contains data from more than one block. In parallel to the DTC access, data from single address is fetched from level-1 cache. In case of a hit in the DTC the entire burst of data is returned .In case there was a miss in the DTC, only single access is serviced by level-1 data cache.

Store instruction presents a major difficulty to the data trace cache. The first problem is locating specific addresses inside the DTC. Since the index function is the XOR value of the address inside the DTC. Since index function is XOR value of address, we must keep look up table to be able to locate single address for store operation. Once locating all the trace cache line that contain the value need to be updated all these lines must be updated .Updating several blocks require many lookups, resulting in long store latencies.

2.2 Approach 2: Application specific data trace cache

Cache size requirements are constantly increasing in an effort to reduce average memory latency. For example, the Intel Itanium 2 processor has a 32KB L1, 256KB L2 and a 4MB L3 [12]. To keep cache area costs low, efforts are needed to utilize caches more efficiently. This approach addresses the issue of improving cache performance by exploiting the unique memory behavior of a large class of applications keeping cache area costs low. Improving the memory system performance of any application is largely founded on the discovery and exploitation of reference locality. Reference locality, though present, may in fact be difficult to detect. The approach is to take in to consideration the application data structures for an important group of embedded kernels: those which access large rooted tree data structures. This examination reveals a form of reference locality that can be exploited using a small cache designed specifically for accesses to this rooted tree [13]. An access to the tree involves a traversal starting from the root node and proceeding to a leaf. Such a traversal involves access to a single node at each level generating a relatively small number of memory accesses relative to the size of the tree. Also the nodes closer to the root are accessed more often than those closer to the leaves. The fact that only one node at each level is accessed is, exploited in generating the layout of the trees in memory. The resulting cache design is a data trace cache (DTC). The DTC design utilizes a flexible placement policy, distinguished from the fixed, application independent placement policies of traditional cache architectures. Only memory accesses to the application tree data structures are processed by the DTC. Non-tree data structure references are handled by a conventional memory hierarchy.

Additionally, in certain high performance architectures such as network processors, a cache hierarchy is normally absent. This is due to the general belief that packet data processing exhibits little reference locality. This is compounded by the fact that caches produce good average case behavior, while network processors require worst case behavior to be acceptable [14]. But using the DTC to cache the application data structure alone (e.g. the routing table or packet classification tables) can yield benefits.

3. MODIFICATIONS TO APPROACH 2: (APPLICATION SPECIFIC DATA TRACE CACHE)

Various modifications to the above devised approach have been suggested. Like in the given approach simple breadth first remapping has been used. Also the Partitioning and Placement of the data is being guided solely by the runtime indexing simplicity requirement of the given cache architecture. The approach we are suggesting is to use the two semiautomatic tools i.e. ccmorph and ccmalloc to implement the remapping as well as partitioning and placement of the data. These tools will significantly reduce the level of programming effort, knowledge, and architectural familiarity [15]. Ccmorph and ccmalloc— use cache-conscious reorganization and cache conscious allocation strategies to produce cache conscious pointer structure layouts. ccmorph is a transparent tree reorganizer that utilizes topology information to cluster and color the structure. ccmalloc is a cache-conscious heap allocator that attempts to co-locate contemporaneously accessed data elements in the same physical cache block.

3.1 Comparison of Breadth First and Depth First Remapping:

Here we are comparing the miss rates when we are remapping the tree data structure either in breadth first order or in the depth first order.

In case the tree data structure is remapped in the depth first order the miss rate for the remapped data can be calculated as:

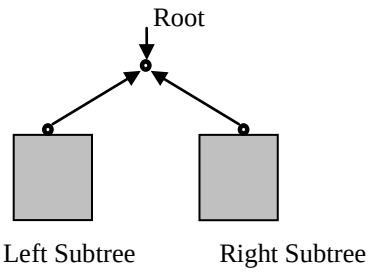

Fig 2. Depth First Remapping:

Suppose we are walking this tree left-to-right as in Fig. 2. Then 1 and 2 are on the same line and 3 is on a different one. We will get a miss when we are coming to the root of the tree i.e. node 1. 1 and 2 are on the same cache line, so

we are not going to get a miss when we access 2. We are also going to get a miss when we access 3, since it's on a different cache line. We can count the precise number of misses:

$$T_{LR}(d) = 1 + (T(d-1) - 1) + T(d-1) = 2T(d-1)$$

The first term corresponds to the initial miss, terms two and three correspond to the number of misses in the left and right subtree, respectively. With an appropriate initial condition, this yields $T(d) = 2^d$.

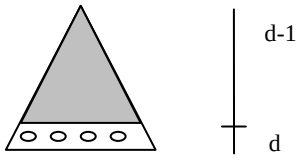

Fig 3. Breadth First Remapping:

We can compute the number of misses created after a BF allocation inductively.

When allocation is breadth-first, every other node at the bottom level causes a miss. For a binary tree Fig. 3, the size of the bottom level is the same half the size of the tree, so

$$T(d) = T(d-1) + 2^d,$$

which means that $T(d) = 2^d$. So the number of misses is the same as in the case of depth-first allocation.

Above results show that just changing the remapping strategy from breadth first to depth first is not going to effect the performance in any way. So, we need to think of some other remapping method. One more method that can be used is the subtree remapping strategy. This method involves the benefits of both breadth first as well as depth first methods. This same method is being used by ccmorph.

3.2 Semiautomatic tools: ccmorph and ccmalloc

Ccmorph reorganizes a data structure using clustering and coloring. Clustering attempts to pack in a cache block data structure elements likely to be accessed contemporaneously. Coloring is used because caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache line without incurring conflict misses. Ccmorph operate on trees and the programmer must supply it a pointer to the root of a data structure, a function to traverse the structure, and cache parameters. The other function, ccmalloc, attempts to locate the new data item in the same cache block as the existing item. It takes an additional parameter that point to an existing data structure element likely to be accessed contemporaneously.

3.3 Using ccmorph and ccmalloc for Data Trace Cache Implementation:

3.3.1 Ccmalloc:

Ccmalloc is used for cache-conscious allocation of memory. The main difference from a regular malloc is that ccmalloc takes as an extra argument, a pointer to some data structure that is likely to be referenced close (in time) to the newly allocated structure. ccmalloc attempts to allocate the new data in the same cache conscious(cc) -block as the data structure pointed at by the argument pointer. Suppose the parents and their children in a binary tree are attempted to be allocated together, then ccmalloc invokes calls to the standard malloc in two cases; when allocating a new cc-block or when the size of the data structure is larger than the cc-block. Otherwise, if called with a pointer to an already allocated structure, the new structure is put in empty slot in the cc-block right after that structure. When no proper area is found, ordinary malloc is called with the cc-block size.

```
#ifdef CCMALLOC
child = ccmalloc(sizeof(struct node), parent));
#else
child = malloc(sizeof(struct node));
#endif
```

The block diagram of how the application is to be adapted to work with our Data trace cache system is:

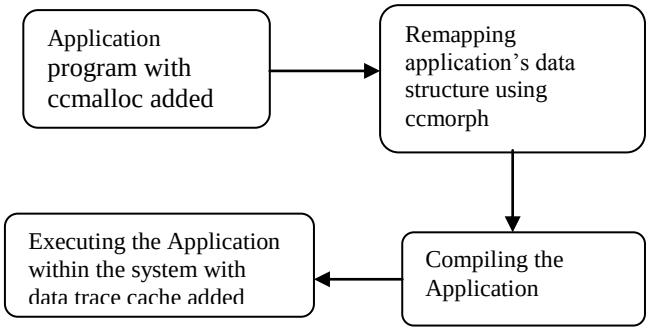

Fig. 4 Data Trace Cache System

3.3.2 Benefits over the previous approach:

Major benefit of our approach over the previous approach is that using cache conscious technique can help us to adapt our system to the specific needs of the application as well as underlying system by choosing the cache-conscious block size, or *cc-block size*, according to its data structures and to the specific cache line size of the system. Also the cc-block size can be set dynamically in software, independently of the hardware cache line size. This means that even though the hardware cache line is smaller than the used data structures, ccmalloc can take advantage of co-allocating data structures, and can be varied depending on the size of the data structures the programmer wants to co-allocate. Another benefit is that we use subtree clustering for remapping; as in binary subtree clustering, for a series of random tree searches, the probability of accessing either child of a node is 1/2. With *k* nodes in a subtree clustered in a cache block, the expected number of accesses to the block is the height of the subtree, i.e. $log_2(k+1)$, which is greater than 2 for $k > 3$. Consider the alternative of a depth-first clustering scheme, in which the *k* nodes in a block form a single parent-child-grandchild-... chain. In this case, the expected number of accesses to the block is:

$$1 + 1*\tfrac{1}{2} + 1*\tfrac{1}{2^2} + \ldots 1 + \tfrac{1}{2^{k-1}} = 2*(1 - (\tfrac{1}{2})^k) \le 2$$

So, subtree clustering improves the probability of access of contagiously placed nodes.

4. CONCLUSION

Application specific data trace caches with static remapping coupled with partitioning and placement functions can bring significant reductions in the total number of misses. The DTC demonstrates this opportunity by achieving significant reductions in miss rates for a variety of applications based on tree data structures used in domains ranging from packet processing applications to databases.

While techniques such as clustering and coloring can improve the spatial and temporal locality of pointer-based data structures, applying them to existing codes may require considerable effort. These cache-conscious techniques have been packaged into easy-to-use tools. The data structure reorganizer, ccmorph, and cache-conscious memory allocator, ccmalloc, greatly reduce the programming effort and application knowledge required to improve cache performance. These cache-conscious structure layout tools are fairly automated, they still require programmer assistance to identify tree-structures that can be moved, and suitable candidates for cache block co-location.

4.1 Future Work

The approach we have explained above considers applications that use a single data structure. The same approach can be extended to applications that use multiple data structures. Even though in the application, often references to one structure predominate, this technique can be applied to each structure in turn to improve its performance. Furthermore interactions among different structures can also be taken into consideration.

This technique has been studied in the context of uniprocessor systems. In multiprocessor systems, we need to consider the data access patterns of different processors individually as cache performance depends on whether the data items are accessed by same processor or by different processors. Because, co-locating the data elements without considering the underlying processor could exacerbate false-sharing.

5. REFERENCES

[1] Krishna V. Palem and Rodric M. Rabbah . Design Space Exploration and Optimization of Embedded Cache Systems via a Compiler. In ACM Transactions in Embedded Computing Systems.

[2] Eric Rotenberg, Steve Bennett, James E. Smith 1996. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching . In the Proceedings of the 29th Annual International Symposium on Microarchitecture.

[3] Chen Ding, Ken Kennedy 1999. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. SIGPLAN ‘99 (PLDI) Atlanta, GA,

[4] D. Callahan, S. Carr, and K. Kennedy 1990. Improving register allocation for subscripted variables. In Proceedings of the SIGPLAN ‘90 Conference on Programming Language Design and Implementation, White Plains, NY.

[5] M. E. Wolf and M. Lam 1991. A data locality optimizing algorithm. In Proceedings of the SIGPLAN ’91 Conference on Programming Language Design and Implementation, Toronto, Canada.

[6] W. Abu-Sufah, D. Kuck, and D. Lawrie 1981. On the performance enhancement of paging systems through program analysis and transformations. IEEE Transactions on computers, C-30(5):341-356.

[7] D. Gannon, W. Jalby, and K. Gallivan 1987. Strategies for cache and local memory management by global program transformations. In Proceedings of the First International Conference on Supercomputing. Springer-Verlag, Athens, Greece.

[8] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems

[9] M. Cierniak and W. Li 1995. Unifying data and control transformations for distributed share d-memory machines. In Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation, La Jolla.

[10] R. M. Rabbah and K. V. Palem 2003. Data remapping for design space optimization of embedded memory systems. ACM Transactions in Embedded Computing Systems.

[11] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999.Cache-conscious structure definition. In Proceedings of the ACM SIGPLAN’99 Conference on Programming Language Design and Implementation.

[12] Intel Itanium 2 Processor Hardware Developer’s Manual, 2002.

[13] Subramanian Ramaswamy, Jaswanth Sreeram, Sudhakar Yalamanchili, K. V. Palem 2005. Data Trace Cache: An Application Specific Cache Architecture . In MEDEA 2005 workshop in conjunction with PACT 2005.

[14] Intel IXP2800 Network Processor Hardware Reference Manual, 2002

[15] Trishul M. Chilimbi, Mark D. Hill, James R. Larus 1999. Cache-Conscious Structure Layout . . In Proceedings of the SIGPLAN ’99 Conference on Programming Language Design and Implementation, Atlanta.