Data Trace Caches

CS/ECE 752: Advanced Computer Architecture I Project Report

Saisanthosh Balakrishnan  Pradeepsunder Ganesh
{santhosh, pradeep}@cae.wisc.edu
Department of Electrical and Computer Engineering
University of Wisconsin, Madison

Contents

1 Acknowledgements 5

2 Introduction 5
  2.1 Motivation ................................................................. 5
  2.2 Background ............................................................. 5
  2.3 Approach ................................................................. 6

3 Linked Data Structures 7
  3.1 Definition ............................................................... 7
  3.2 Data Access Pattern in LDS ........................................... 7

4 Data Trace Cache 9
  4.1 Design ................................................................. 9
  4.2 How is trace created ? .................................................. 10
  4.3 Lookup Table .......................................................... 10
  4.4 Read operation ......................................................... 10
  4.5 Write operation ....................................................... 11

5 Implementation 11
  5.1 SimpleScalar .......................................................... 11
  5.2 Changes Done To SimpleScalar ....................................... 11

6 Simulations 16
CONTENTS

6.1 L2 References ................................................................. 17
6.2 Replacement Rate ............................................................ 18
6.3 L1 Miss Rate ....................................................................... 19
6.4 Unique Patterns in the Data Trace Cache ......................... 20
6.5 Improvement in L1 Hit Rate ................................................ 21
6.6 Data Trace Cache Miss Rate ............................................... 22

A Data Trace Cache Source Code ........................................... 25
A.1 dtcache.h – Data trace cache definition .............................. 25
A.2 dtcache.c – Data trace cache functions ............................... 28
A.3 dtcache_params.h – Data trace cache parameter ................. 36
A.4 dt_l1_link.c – Links sim-cache and Data trace cache .......... 37

B Detailed Simulation Data ..................................................... 43
LIST OF TABLES

List of Tables

1  Pointer Load Contribution ......................................................... 8
2  Design Parameters of DTC ............................................................. 12
3  Modifications to SimpleScalar 2.0 .................................................... 13
4  Function Descriptions ................................................................. 14
5  Benchmarks ............................................................................... 16
6  L2 References With Data Trace Cache ............................................. 43
7  Data Trace Cache Miss Rates ......................................................... 43
8  L1 Miss Rate ............................................................................... 44
9  Unique Patterns in the Data trace Cache ......................................... 44
10 Replacement Rate of Data Trace Cache ............................................ 45
11 Improvement in L1 Hit Rate because of Data Trace Cache Hits ............ 45

List of Figures

1  Linked Data Structure, Program and Data access Pattern ....................... 8
2  Data Trace Cache Line ..................................................................... 9
3  Data Trace Cache Block .................................................................. 9
4  Placement of Data Trace Cache ....................................................... 10
5  Structure of Lookup Table .............................................................. 11
6  L2 References .............................................................................. 18
7  Replacement Rate .......................................................................... 19
8  L1 Miss Rate .................................................................................. 20
9  Unique Patterns in DTC ................................................................... 21
10 Improvement in L1 Hit Rate ............................................................ 22
11 DTC Miss rate .............................................................................. 23

List of Algorithms

1  new_dttrace – Create a new trace in the Data trace cache and add ............ 14
2  lookup_dtcache – Searches the Data Trace Cache for a trace line with the given address ........................................... 14
3  merge – Algorithm to merge the newblock in a Traceline ....................... 15
4  buildtrace - Building of a Trace given an Address A and data of size nBytes  


1 ACKNOWLEDGEMENTS

Abstract

The memory hierarchy attempts to bridge the increasing gap between processor and memory speeds. However traditional caches do not always capture a program's reference locality, causing the processor to stall on each cache miss. Hence, the memory hierarchy is a bottleneck in achieving high instructions per cycle.

We propose the Data Trace Cache (DTC), a low latency mechanism for supplying the processor with several blocks of data that may not be physically contiguous, but have been sequentially accessed during past execution. This scheme potentially offers better performance than conventional caches because access to non-contiguous memory addresses can sometimes be fulfilled with one memory request, and supplying blocks in the order encountered during execution may represent future access patterns more accurately than traditional address-based cache organizations, especially in linked data structures.

1 Acknowledgements

We thank Prof. Gurindar S. Sohi and TA James Nugent for guiding us in this project.

2 Introduction

2.1 Motivation

The gap between processor and main memory cycle times continues to increase. Traditional memory hierarchies mitigate this problem with caches which exploit reference locality. Unfortunately, conflict and capacity misses prevent caches from perfectly capturing a processor's reference patterns. Even on cache hits, upper layers of the memory hierarchy, or the processor will stall due to the cache access latency, an effect more pronounced when running programs that use some form of a Linked data structures.

2.2 Background

Research into the behavior of application memory access patterns can be split roughly along two axes. The first axis is when the suggested methods are applied: during coding, which is application development and compilation, or during execution. The second axis is what form the data structures of the program take, array based or pointer based Linked Data Structures.

Array based applications, which are usually scientific, lend themselves to performance tuning during coding. This is because they are usually relatively amiable to the control structure reorganization techniques discussed in class. These techniques include loop manipulation, loop interchange and tiling all using a core of dependence analysis. They are a forward solution to the problem of memory usage in that the program is adapted to the memory hierarchy and not the other way around. This is the best solution as it can be the cleanest in execution if not in code. However, it can be very difficult to recognize the opportunities and manage the transformations - even for a compiler. These types of programs and techniques are not the focus of this paper.

Linked Data Structures, LDS, which are based on pointers, such as trees, linked lists and hash tables lack even the partially developed theory available for array based data structures except for a few special
cases and heuristics. [2] This condition is especially true if the traversal of the LDS is dynamic. Dynamic traversal is a hallmark of many LDS algorithms and is rarer among array based applications.

LDS algorithms often exhibit a behavior called pointer chasing. This is the term given to indirect loads that use the result of a previous load as their base address. In some applications, where little work is done at each node, pointer chasing can dominate the runtime of the algorithm because of cache misses. This is because when the nodes were allocated they were placed wherever the system wished to put them and not in a position that would work well for the desired, often unknown, traversal of the LDS. The depth of this chain of indirection can vary; in the case of linked lists it can be very high indeed, up to the length of the list.

When the control structure of the algorithm, and hence memory access pattern, can not be restructured prefetching is often used to minimize cache misses. A simple example in array based programs is when there is a linear stride through the array. It should be possible to issue the loads ahead of time so that they are completed when the program really wants them. There are two ways that prefetching can be obtained, manually during coding and dynamically during execution.

Manual prefetching usually takes the form of an ISA level instruction, that can not cause any exceptions, that is inserted into the instruction stream before the related load. For optimal benefit it should be issued as far in advance as the memory system latency. This style of prefetching must be orchestrated by the compiler or a human optimizer at coding time and is useful only when there is significant understanding of the memory usage pattern before execution. An example of this is the support provided in the IA-64 discussed in class.

Dynamic prefetching can often be implemented in hardware. A relatively simple prefetch engine, even in hardware, can detect the arithmetic repetition of linear array accesses and start issuing speculative loads. More complicated array access patterns, or LDS structures can confuse all but a very sophisticated prefetch engines.

LDS programs present particular problems for prefetching because of lack of arithmetic relation between node addresses. In some case the traversal pattern is simple enough, such as depth-first that they can be inserted at coding time and even somewhat recognized by hardware prefetch engines. [2] The difficulties stem directly from the pointer chasing.

An orthogonal approach to alleviate this problem is to “memorize” the data structure traversals when they are accessed the first time, and providing the L1 cache with the trace of non-contiguous data during a miss to the same address. This significantly reduces the L1 miss rate and the number of L2 references.

2.3 Approach

Our Work builds on the Instruction Trace Cache (ITC) [3] - a structure that captures dynamic instruction sequences. The ITC supplements the instruction cache by providing the decoder in a microprocessor with merged and aligned basic blocks that may be noncontiguous, but have been executed sequentially in the past. The ITC lowers instruction fetch latency by amortizing the instruction fetch cost over the multiple instructions stored in an executed trace. The selection of a trace is based on its starting address and outcomes of the branch instructions included in the trace. These outcomes are compared against branch prediction information provided by the processor. Simulations of SPEC benchmarks have shown that a 4 KB ITC can increase a processor’s IPC by an average of 28%, compared to the conventional sequential
fetching.

3 Linked Data Structures

In this section we briefly talk about Linked Data Structures or LDS, and the contribution of pointer loads to miss rate of L1 cache.

3.1 Definition

Linked Data Structures (an example program, structure and data access pattern is shown in Figure 1) are common in many applications and their importance is growing with the spread of object-oriented programming. The popularity of LDS stems from their flexibility, and not their performance. LDS access, entail chains of data dependent loads that serialize address generation and memory access. In traversing an LDS, these loads often form the program’s critical path. Consequently, when they miss in the cache, they can severely limit parallelism and degrade performance.

LDS-specific memory behavior can be summarized by examining the load instructions that access LDS elements, or pointer loads. A pointer load is a load whose input base address was produced by another load instruction. This definition encompasses LDS accesses, and distinguished them from stack and array loads, whose addresses are computed arithmetically, and loads that use addresses produced by a means other than an indirection.

Table 1 shows the number of loads and the data miss rate for a 32KB, 2-way, 32B line data cache. Pointer loads represent a large fraction of all loads in the benchmark programs and contribute a disproportionately larger fraction of the cache misses, accounting for nearly all misses in many of the programs.

Linked Data structures (LDS) are pointer based data structures, in which a structure contains a pointer to another instance of the same structure (or perhaps a different structure). The simplest example is a singly-linked list, which we can describe as follows.

3.2 Data Access Pattern in LDS

In Figure 2, the program for $i = 1$ root node is accessed. In the iterations that follow, the nodes accessed in sequence are nodes pointed to by the next pointer of the current node. There is a good possibility that these nodes are not contiguous in memory. Therefore, there could be a miss in the L1 cache. If data is present in L2 cache it is brought from L2 cache to L1 cache after some latency. This process goes on till the last node is reached. We observe from the above access pattern that nodes can be prefetched from the L2 cache only after its predecessor node has been fetched and so on. Thus, LDS imposes a recurrent serial load behavior.
struct node {
    char *item;
    struct node *next;
}

for (i = 0; i < 10; i++)
    node = head;
    while (node)
        sim += node->data;
        node = node->next;
    endwhile
eンドfor

Figure 1: Linked Data Structure, Program and Data access Pattern

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>octree</td>
<td>4K bodies</td>
<td>720 KB</td>
<td>866 M</td>
<td>29.1%</td>
<td>16.3%</td>
<td>53.0%</td>
</tr>
<tr>
<td>bisort</td>
<td>binary tree</td>
<td>250000</td>
<td>1535 KB</td>
<td>625 M</td>
<td>15.4%</td>
<td>49.1%</td>
<td>99.4%</td>
</tr>
<tr>
<td>em3d</td>
<td>lists</td>
<td>2000 nodes</td>
<td>1670 KB</td>
<td>60 M</td>
<td>23.5%</td>
<td>59.2%</td>
<td>81.5%</td>
</tr>
<tr>
<td>health</td>
<td>quadtree</td>
<td>5 levels</td>
<td>925 KB</td>
<td>169 M</td>
<td>36.2%</td>
<td>81.1%</td>
<td>98.3%</td>
</tr>
<tr>
<td>mst</td>
<td>array of nodes</td>
<td>1024 nodes</td>
<td>20 KB</td>
<td>256 M</td>
<td>14.6%</td>
<td>41.3%</td>
<td>83.5%</td>
</tr>
<tr>
<td>perimeter</td>
<td>quadtree</td>
<td>4Kx4K</td>
<td>6445 KB</td>
<td>1619 M</td>
<td>17.1%</td>
<td>16.2%</td>
<td>99.7%</td>
</tr>
<tr>
<td>power</td>
<td>multiway tree</td>
<td>10000 nodes</td>
<td>313 KB</td>
<td>791 M</td>
<td>18.9%</td>
<td>12.2%</td>
<td>91.6%</td>
</tr>
<tr>
<td>treeadd</td>
<td>binary tree</td>
<td>1M nodes</td>
<td>12300 KB</td>
<td>196 M</td>
<td>20.6%</td>
<td>15.8%</td>
<td>97.2%</td>
</tr>
<tr>
<td>tsp</td>
<td>binary tree</td>
<td>100000 nodes</td>
<td>5120 KB</td>
<td>338 M</td>
<td>9.4%</td>
<td>74.0%</td>
<td>99.8%</td>
</tr>
<tr>
<td>voronoi</td>
<td>binary tree</td>
<td>60000 points</td>
<td>1100 KB</td>
<td>333 M</td>
<td>14.3%</td>
<td>71.2%</td>
<td>41.1%</td>
</tr>
</tbody>
</table>

Table 1: Pointer Load Contribution
4 Data Trace Cache

The Data trace cache is a low latency mechanism for supplying the processor with non-contiguous streams of data which have been accessed sequentially during earlier part of the execution. The design aspects of the Data Trace Cache are as follows:

4.1 Design

- The Data trace cache is defined by the parameters listed in Table 2.
- A Data trace cache can have atmost $m$ non-contiguous blocks with a maximum of $n$ bytes in each trace line as shown in Figure 2.

![Instruction Trace Cache](image)

![Data Trace Cache](image)

Figure 2: Data Trace Cache Line

- Each block (Figure 3 in the Data trace cache line, contains the start and end addresses, along with other required attribute bits.

![DTC Block](image)

Figure 3: Data Trace Cache Block

- Data Trace Cache is accessed in parallel with L2 and is shown in Figure 4. The algorithm for looking up a trace cache and inserting a new trace line are explained in Algorithm 2 and Algorithm 1.
4.2 How is trace created?

When L1 cache misses, and the DTC also misses, the building of a new trace starts. All following loads that miss in the L1 cache are compacted into the trace line buffer. Contiguous data loads are merged and compacted into the same blocks in the trace buffer. Non-contiguous data loads are stored as separate blocks in the trace line. When the number of blocks reaches $m$ or when the total number of bytes in the buffer reaches $n$, a new trace line is created and the buffer added. The lookup tables are updated accordingly.

4.3 Lookup Table

The DTC maintains a lookup table (Figure 5) for each line in the DTC. The lookup table consists of a maximum end address and the minimum start address and a valid bit for each line in the DTC. The purpose of the Look up Table is to reduce the DTC lookup latency and to maintain the coherency of data in the DTC.

To check whether a given address $A$ is in Trace Cache, all valid entries in the lookup table are searched using the minimum starting and maximum ending address, to determine as to which trace lines may have that address.

The particular address is searched in the trace line in the Data trace Cache.

4.4 Read operation

During a Read miss in the L1 cache the address is sent to the Data trace cache, if it hits in the data trace cache, the entire trace line is loaded into the L1 cache. However, if it misses in the Data trace cache, a new trace line is allocated for the address, and all the succeeding loads that miss in the L1 cache and Data trace cache are compacted into the trace line buffer. The address that missed both in L1 and Data trace cache is brought from L2 cache.
4.5 Write operation

The Data trace cache follows a write-through policy. Therefore, when there a Write hit in the L1 cache, the data is written to the appropriate trace cache lines. However, if the write misses in the L1 cache, since the SimpleScalar toolset’s cache model follows write-allocate policy, the data is written in the L2 cache and brought back into L1 cache.

5 Implementation

5.1 SimpleScalar

SimpleScalar [1] tool set performs fast, flexible and accurate simulation of modern processors that implement the SimpleScalar architecture (a close derivative of MIPS architecture). The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators. The advantages of the tools are high flexibility, portability, extensibility and performance. The five execution-driven processor simulators are sim-fast, sim-safe, sim-cache, sim-profile, sim-outorder. Of these, sim-cache is ideal for fast simulation of caches if the effect of cache performance on execution time is not needed. Since our main goal was to show that, the DTC was effective in reducing number of L1 Misses and L2 References we used sim-cache.c for our execution-driven simulation. sim-cache simulator accepts command-line arguments for configuring a level-one and level-two data and instruction caches. The cache parameters supported include number of sets, block size, associativity and replacement policy. The DTC simulator (explained in detail in the following subsection) adds a command-line argument for specifying the parameters of DTC. Table 2 describes the parameters in detail.

5.2 Changes Done To SimpleScalar

The Data Trace Cache parameters are present as hash defined values in the file dtcache.h, and they can be modified to obtain various Data Trace Cache configuration.
Table 2: Design Parameters of DTC

Read Operation

__READ_CACHE macro. The Read Operation calls the __READ_CACHE(addr,sizeofdata) macro which is defined in the sim-cache.c. This macro was modified to call the push_addr() function which in turn called the function push_addr with the addr and the size of the data being read.

dll_access_fn, This function is present in sim-cache.c. When the Read miss occurs in the L1 cache the dll_access_fn is called to service the miss. This function then gets the data from the memory lower in the hierarchy than L1. This function was modified to log a global variable L1miss as TRUE whenever the miss occurs, which is required to enable the logging of the traces in the push_addr function. Also the dll_access function was modified to call lookup_dtcache to check whether the trace with this address is present.

push_addr when this function is called each time it checks whether it is a miss in L1 cache if it is not then it does not do anything but just returns but if it is a miss in the L1 cache this function serves the purpose of building the traces from the L1 misses and when the trace line gets completed passes the trace line to the data trace cache. This function is present in the file dt_l1_link.c and maintains the traces in the buffer till the trace can be transferred to the data trace cache. So every time the Read miss happens in L1 the push_addr is called, with the address which caused the miss and the bytes that are to be read. This address and the bytes gets inserted into the current data trace line buffer and also the total number of bytes in the current line gets incremented by the number of bytes added and if the data is contiguous it is merged and compacted into the already existing block by calling the function mergeendstart[3]. This function is present in the file dt_l1_link.c. If the data is non-contiguous then the data is inserted as a new block in the trace line and the total number of blocks in the current trace line is incremented by one this is done by the incrementbytes function present in the dt_l1_link.c file. When the total number of bytes in the buffer is greater than the Nbytes or when the total number of non-contiguous blocks becomes greater than the M then the push_addr function inserts the trace line from the buffer into the data trace cache by calling the new_dttcache function and starts building the new trace on the next cache miss.

Write Operation

During a write operation, when there is a write hit in the L1 cache, the Data Trace Cache has to be updated to maintain coherency. So the dt_line_invalidate function which searches the lookup table to identify which trace lines the address is present and then updates that particular block in all the trace
5 IMPLEMENTATION

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim-cache.c</td>
<td>Modify d11_access_fn to call lookup_dtcache when there is a L1 cache miss, added</td>
</tr>
<tr>
<td></td>
<td>instrumentation code for measuring L2 references with and without Data Trace</td>
</tr>
<tr>
<td></td>
<td>Cache.</td>
</tr>
<tr>
<td>dtcache_params.h</td>
<td>Defines the size of Line size, Number of sets, Number of associativity and Number</td>
</tr>
<tr>
<td></td>
<td>of Blocks.</td>
</tr>
<tr>
<td>dtcache.h</td>
<td>Definitions of Data trace cache functions and the structure.</td>
</tr>
<tr>
<td>dtcache.c</td>
<td>Implement Data trace cache functions and instrumentation code.</td>
</tr>
<tr>
<td>dt_l1_link.c</td>
<td>Functions for compaction of serial load traces which calls create_dtcache</td>
</tr>
<tr>
<td>Makefile</td>
<td>Added “dtcache.c”, “dtcache.h”, “dtcache_params.h”</td>
</tr>
</tbody>
</table>

Table 3: Modifications to SimpleScalar 2.0

lines. The dt_line Invalidate function calls the write Invalidate function which is also present in dtcache.c which does the actual update of the trace line with the new data. If the write is a miss, then the cache does a write-allocate to L2, i.e., it writes to L2 and then reads L1 which misses and hence the newly updated data is brought from the L2 to L1. So the push_addr function is called when the write miss occurs. This modification has been done in the cache.c. So when the push_addr is called the miss is logged in the buffer maintaining the current trace line.

Data Trace Cache Creation and Trace Line Insertion

The function create_dtcache is used to create the Data Trace Cache. The parameters for the Data Trace Cache is determined in the hash defined values in the dtcache_params.h. The create_dtcache function was called in the function sim-check options which is present in sim-cache.c. The function lookup_dtcache (Algorithm 2) is used determine whether the particular address is present in the Data Trace Cache. The d11_access_fn in sim-cache.c which is the L1 cache miss handler was modified to include lookup_dtcache function to check whether the address that missed in the L1 cache is present in the Data Trace Cache. If the address is present in the Data Trace Cache the entire trace line gets loaded into the L1 cache and the load from the L2 cache is bypassed. The function new_dttrace (Algorithm 1) is used to insert a trace line that has been built in the buffer into the Data Trace Cache. The push_addr function in dt_l1_link.c was modified to call this function when the trace line became complete in the buffer. If the Data Trace Cache is full and a new line needs to be created by bumping an already existing line, the function find_new_trace_line, which is present in the file dtcache.c is called, which depending on the replacement policy of the Data Trace Cache bumps the line accordingly and new trace line is inserted in its place.

The functions scan_dttrace and print_trace_line were used for debug purpose, and the function dt_stats was used to print the simulation results pertaining to the Data Trace Cache. These functions are also present in the dtcache.c file.

Tables 3 and 4 tabulates the changes done to the SimpleScalar toolset and functions added to it.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_dtcache</td>
<td>Creates a Data trace cache with the specified parameters such as Number of Sets, Associativity, Block Size, Number of Blocks</td>
</tr>
<tr>
<td>lookup_dtcache</td>
<td>Lookup the Data trace cache for a specified starting address, the function returns the trace line if it is a DTCA_CACHE_BLK_HIT</td>
</tr>
<tr>
<td>find_new_trace_line</td>
<td>Finds a new trace line (set and block) for the given address.</td>
</tr>
<tr>
<td>new_dtrace</td>
<td>Inserts a new Data trace line into the Data trace cache. The trace line contains at least $m$ non-contiguous blocks or $n$ bytes.</td>
</tr>
<tr>
<td>write_invalidate</td>
<td>Invalidates all the Data trace lines that contain an address $A$ during writes.</td>
</tr>
<tr>
<td>push_addr</td>
<td>Builds the trace line and inserts the trace line into the Data Trace Cache.</td>
</tr>
<tr>
<td>merge_endstart</td>
<td>Merges and compacts the contiguous data into the same block.</td>
</tr>
</tbody>
</table>

Table 4: Function Descriptions

**Algorithm 1** `new_dtrace` – Create a new trace in the Data trace cache and add

1: Let $A$ be starting address of the 1st block in the buffer.
2: Let `set` be the Set Number of the trace and `index` the block in that set.
3: Minimum Start Address ← Minimum Start Address of all the blocks present in the buffer.
4: Maximum End Address ← Maximum End Address of all the blocks present in the buffer.
5: Insert the Minimum Start and Maximum End Addresses into the lookup table at the appropriate position found by the tag.
6: Create a new trace line and copy the $m$ blocks or $n$ bytes from the buffer to the newly created trace line.
7: Set the required attribute bits of the trace line.

**Algorithm 2** `lookup_dtcache` – Searches the Data Trace Cache for a trace line with the given address

1: Let $A$ be the starting address of the data that has missed in L1 Cache.
2: Search the Lookup table parallelly for address $A$. The search function returns the entries that have address $A$ within the minimum start address and maximum end address range.
3: Search for address $A$ in the trace lines, if DTCA\_CACHE\_BLK\_HIT and DTCA\_CACHE\_BLK\_VALID load the data into the L1 cache.
Algorithm 3 merge - Algorithm to merge the newblock in a Traceline
1: for Current block = First block of current trace line to the last block of current traceline do
2:   if Current block.startaddress < newblock.startaddress and current block.endaddress > newblock.endaddress then
3:     Compact the trace line
4:   end if
5:   if New block.endaddress > current block.endaddress and new block.start address < current block.startaddress then
6:     Replace current block with the new block
7:   end if
8:   if New block.Startaddress = Current block.Startaddress or New block.Startaddress = Current block.Startaddress + 1 then
9:     merge the new and current block
10:    Current block.Endaddress = New block.Endaddress
11:   Discard the newblock
12: end if
13: if Newblock.endaddress == current block.startaddress or newblock.endaddress +1 == current block.startaddress then
14:   Merge the new block with the current block
15:   Current block.startaddress = newblock.startaddress
16:   Discard the new block
17: end if
18: end for
19: if no condition satisfies return 0 to calling program

Algorithm 4 buildtrace - Building of a Trace given an Address A and data of size nBytes
1: if L1 Miss occurs and address misses in DTC also then
2:   A new block is created with given start address A
3:   call merge
4:   if merge returns True then
5:     Return
6:   else
7:     if total bytes in the current trace line > Maximum Bytes per trace line allowed or total blocks in current trace line > Maximum number of blocks allowed per trace line then
8:       Call new dttrace to insert the trace line in to the Data Trace Cache
9:       Reset the values for total bytes and total blocks
10: end if
11: end if
12: end if
6 Simulations

To measure the effectiveness of our proposed Data trace cache, we ran our execution-driven simulator on 11 different benchmark programs. These programs were collected from:

- Olden Benchmark Suite: These benchmarks were originally developed in the context of studying pointer chasing on parallel machines. The authors of [2] stripped these programs down to standard sequential code and they have been used for sequential pointer-chasing experimentation since.
- Todd-Austin's pointer Intensive Benchmark Suite

The benchmarks used, description and input sets for the simulator are in Table 5.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>Simulate the evolution of the particles in a gravitational system. This is divided into three steps: represent the particles in a tree, calculate new acceleration for the particles by walking through the tree, calculate the new position for the particles.</td>
<td>300 nodes.</td>
</tr>
<tr>
<td>ft</td>
<td>Minimum-span calculation, lots of dynamic storage allocation and deallocation throughout the life of program, very little pointer arithmetic.</td>
<td>1000 nodes, 2000 edges.</td>
</tr>
<tr>
<td>ks</td>
<td>Kernighan-Schreiber graph partitioning tool, lots of pointer and array dereferencing and arithmetic, some dynamic storage allocation, no explicit dynamic storage deallocation.</td>
<td>KL-2.in</td>
</tr>
<tr>
<td>mst</td>
<td>Computes the minimum spanning tree of a graph.</td>
<td>256 nodes</td>
</tr>
<tr>
<td>perimeter</td>
<td>Compute the perimeter of a set of quad-tree encoded raster image.</td>
<td>8 levels of quadtree.</td>
</tr>
<tr>
<td>treeadd</td>
<td>Adds the values in a tree.</td>
<td>128 K nodes.</td>
</tr>
<tr>
<td>voronoi</td>
<td>Compute the Voronoi Diagram of a set of points. This is a classic geometric divide-and-conquer algorithm. The points are stored in a binary tree sorted by x-coordinate. The algorithm computes the Voronoi Diagram for the two subtrees first, and then merges them.</td>
<td>2K nodes.</td>
</tr>
<tr>
<td>bsort</td>
<td>Sort numbers in non-decreasing order using binary sort.</td>
<td>10000 nodes.</td>
</tr>
<tr>
<td>health</td>
<td>Simulate the Columbia Health System using a four-way tree. Each node of the tree represents a hospital, and each node is a list of patients. At each timestep, the patient, once accessed, either treated or sent to parent hospital.</td>
<td>5 levels, 100 iterations.</td>
</tr>
<tr>
<td>simple</td>
<td>Creates a simple linked list and traverses it.</td>
<td>12000 nodes.</td>
</tr>
<tr>
<td>tsp</td>
<td>Computesthe estimate of the best Hamiltonian circuit for the Travelling-salesman problem.</td>
<td>1K cities.</td>
</tr>
</tbody>
</table>

Table 5: Benchmarks
The simulations were conducted on 8 different cache models. The Cache models are:

- 8 KB L1 Data Cache
- 8 KB L1 Data Cache with 4 KB Direct-mapped Data Trace Cache, with 128 bytes line size.
- 8 KB L1 Data Cache with 4 KB 2-Way set-associative Data Trace Cache, with 128 bytes line size.
- 8 KB L1 Data Cache with 4 KB 4-Way set-associative Data Trace cache, with 128 bytes line size.
- 8 KB L1 Data Cache with 4 KB 4-Way set-associative Data Trace Cache, with 64 bytes line size.
- 8 KB L1 Data Cache with 8 KB 4-Way set-associative Data Trace Cache, with 128 bytes line size.
- 16 KB L1 Data Cache.
- 16 KB L1 Data Cache with 4 KB 4-Way set-associative Data Trace Cache, with 128 bytes line size.

There are some limitations in our experimental methodology. First, we implement a write-through policy for the Data trace cache, as write-back policy is complicated to implement because of the variable sized blocks of non-contiguous data. Second, due to the lack of timing information in sim-cache.c, we were unable to model pipelined memory accesses and their effect on performance.

6.1 L2 References

As Figure 6 shows that the Data trace cache successfully lowered the number of L2 references. For all programs, the number of L2 references reduces considerably after adding a Data trace cache. This was achieved by the Data trace cache injected traces of data which the processor subsequently requested, and hit in the L1. Simple is an artificical benchmark, which we created to test the limits of the Data trace cache. It traverses multiple times a linked list of structures that occupy non-contiguous locations in memory. An 8 KB L1 Data cache with 4 KB Data trace cache has lower number of L2 references than a 16 KB L1 Cache. This was not true for Ks and Ft benchmark programs. However, when the two programs outperformed a 16 KB L1 cache when they were simulated on a larger Data trace cache or 8 KB. Thus we infer that the number of data accesses for Ks and Ft are considerably high. This can be seen from Figure 9 on Replacement Rates, where the number of replacements for Ft and Ks were very high compared to other benchmarks.
### 6.2 Replacement Rate

The replacement rates of Data trace caches (Figure 7) are reduced as the size of the Data trace cache increases. Replacement rate of a 64 line size 4 KB Data trace cache was higher than that of a 128 line 4 KB Data trace cache, as a 64 line Data trace cache captures less number of non-contiguous loads and hence has higher probability of being thrown out from the cache. The replacement rate of a 16 KB L1 data cache with 4 KB Data trace cache is lesser than any other cache configuration because of a larger sized L1 cache. The larger sized L1 cache reduces the number of L1 misses, and hence reduces the number of Data trace cache accesses. Interestingly the replacement rate for a larger sized Data trace cache (8 KB) is higher than a 4 KB Data trace cache for most of the benchmarks. The reason for this anomaly being that the data set size used for an 8 KB Data trace cache was smaller and hence was less rigorous in testing the data trace cache’s performance.
6.3 L1 Miss Rate

The L1 miss rate is shown in Figure 8. The L1 miss rate, is reduced after adding Data trace cache. The number of misses is higher for a 16 KB L1 cache than an 8 KB L1 cache with 4 KB of Data trace cache. However, TSP (Travelling Salesman Problem) was an exception to this observation. 16 KB L1 cache with 4KB Data trace cache did not perform very well on some of the benchmarks because of the smaller input set we had used for testing it. An 8 KB L1 cache with 8 KB Data trace cache did not reduce the L1 miss rate considerably because of the saturation on the number of pointer loads. The 8KB Data trace cache is hence oversized for the input set we used. This observation agreed with the informal experimentation that we did on various sizes of the data trace cache. 8 KB Data trace cache with 64 bytes line size, led to a small increase in L1 miss rate than an 128 bytes line size, 8 KB Data trace cache because of lesser number of non-contiguous it captures in one trace line.
6.4 Unique Patterns in the Data Trace Cache

Figure 9 plots the number of unique patterns that are found in the Data trace cache. The number of unique patterns for 16 K L1 cache with 4 K Data trace cache is lesser because of a larger L1 cache. 8K L1 cache with 8K data trace cache has the larger number of unique patterns than a 4K data trace cache because of the larger number of trace lines present. 8K Data trace cache with 64 byte line size has higher number of unique patterns as the number of blocks that can be stored in a trace line is considerably lesser when compared to a 128 byte line size. The associativity does not increase the number of unique patterns in the data trace cache significantly.
6.5 Improvement in L1 Hit Rate

The chart shown in Figure 10 is a plot between the benchmarks and the increase in L1 hits because of loading non-contiguous streams of data from the data trace cache. The Ks benchmark has maximum improvement in the L1 hit rate, because of repeated access of the Linked data structure. Treaddd and Mst do not perform well. But both the programs are known to perform badly on most of prefetching mechanisms proposed because of its interesting data access pattern. The observations made on earlier chart holds good for this chart also.
### Data Trace Cache Miss Rate

The chart in Figure 11 shows the DTC miss rate. The miss rates for 4-way associative caches is lower compared to other Data trace caches because of higher associativity. 16K L1 cache with 4K Data trace cache has higher miss rate than other Data trace caches, but as stated above the data sets weren’t sufficiently large enough to test 16K L1 cache with a Data trace cache rigourously.
Figure 11: DTC Miss rate

References


#ifndef DTCACHE_H
#define DTCACHE_H

#include "ss.h"
#include "memory.h"
#include "misc.h"
#include "dtcache_params.h"

#define DTCACHE_BLK_INVALID -2 /* invalid block */
#define DTCACHE_BLK_VALID 1 /* valid block */
#define DTCACHE_BLK_DIRTY 2 /* dirty block */
#define DTCACHE_HIT 7 /* hit in DTC */
#define DTCACHE_MISS 8 /* Miss in DTC */
#define DT_CACHE_SET(cp, addr) ((addr >> log_base2 (dtusize)) & (cp->sets - 1))
#define DT_COUNTER_TYPE unsigned long long

enum dtcache_policy {DTLRU, DTRandom, DTFifo};
static long long static_fido = 0;

/* structure of a lookup table containing start, end address and other * attribute bits */

struct dtcache_table {
  SS_ADDR_TYPE start_addr, end_addr;
  unsigned char valid;
  unsigned char hit;
};

/* structure of a block that has a tag, status, start and end address * and number of blocks */

struct dtcache_blk {
  SS_ADDR_TYPE tag; /* tag */
  unsigned int status; /* dirty or valid */
  SS_ADDR_TYPE *start_address, *end_address;
  unsigned char no_of_BB; /* number of blocks */
  long long fifo; /* life of the block */
};

/* structure of a data trace cache set, containing the blocks */

struct dtcache_set {
  struct dtcache_blk *blks; /* pointer to a set of blocks */
};

/* structure of a data trace cache */

struct dtcache {
  char name [20]; /* name of the data trace cache */
enum dcache_policy policy;  
int nsets; /* number of sets */  
int assoc; /* associativity */  
int bsize; /* size of each block */  
int m; /* number of blocks */  
struct dcache_set *sets; /* pointer to sets */  
struct dcache_table *dtbl; /* pointer to lookup table */

DT_COUNTER_TYPE hits;  
DT_COUNTER_TYPE unique_hits;  
DT_COUNTER_TYPE misses;  
DT_COUNTER_TYPE writes;  
DT_COUNTER_TYPE ignored_writes;  
DT_COUNTER_TYPE replacements;  
DT_COUNTER_TYPE l2_references_dtc;  
DT_COUNTER_TYPE l2_references;  
DT_COUNTER_TYPE dt_misses;

DT_COUNTER_TYPE bytes_from_l2;  
DT_COUNTER_TYPE bytes_from_dtt;  
DT_COUNTER_TYPE bytes_from_l2_without_dtt;  
DT_COUNTER_TYPE bytes_written;  
DT_COUNTER_TYPE prefetches;

};

struct dcache* create_dcache (char *name,  
enum dcache_policy policy,  
int nsets,  
int assoc,  
int bsize,  
int m);

int lookup_dcache (struct dcache *cp,  
SS_ADDR_TYPE start_addr [],  
SS_ADDR_TYPE end_addr [],  
struct dcache_blk **line,  
unsigned int *no_BB);

int find_new_trace_line (struct dcache *cp,  
SS_ADDR_TYPE addr);

void new_dtttrace (struct dcache *cp,  
SS_ADDR_TYPE start_addr [],  
SS_ADDR_TYPE end_addr [],  
unsigned int no_BB);

void scan_dtttrace (struct dcache *cp,  
SS_ADDR_TYPE addr,  
int nbytes);

void print_trace_line (struct dcache *cp,  
int set,  
int index);
void dt_line_invalidate (struct dtcache *cp,
  SSA_ADDR_TYPE saddr,
  unsigned int nbytes);

void write_invalidate (struct dtcache *cp,
  int set,
  int index, SSA_ADDR_TYPE addr,
  unsigned int nbytes);

int address_subspace (SS_ADDR_TYPE b_saddr,
  SS_ADDR_TYPE b_caddr,
  SSA_ADDR_TYPE saddr,
  SSA_ADDR_TYPE caddr);

#endif
A.2 dcache.c – Data trace cache functions

#include "dtcache.h"
#include <stdlib.h>

/* create_dcache function: creates a data trace cache. The parameters
 * are name, policy such as LRU, FIFO and Random, number of sets,
 * associativity, block size, and number of blocks. The function
 * returns a struct dcache pointer which is a pointer to the Data
 * trace cache that has been created. This pointer is used to access
 * all member of functions of the data trace cache
 */

struct dcache* create_dcache (char *name,
               enum dcache_policy policy,
               int nssets,
               int assoc,
               int bsize,
               int m) {

    struct dcache *cp;
    int i, j;

    /* If nssets is negative the parameter is invalid, terminate */
    if (nssets <= 0)
        fatal("cache size (in sets) \"d\" must be non-zero", nssets);

    /* if associativity if negative the parameter is invalid,
    * terminate */
    if (assoc <= 0)
        fatal("cache associativity \"d\" must be non-zero and positive", assoc);

    /* Allocate memory for a data trace cache */
    cp = (struct dcache *) malloc (1, sizeof (struct dcache));
    if (cp)
        fatal("out of virtual memory");

    /* initialize all the variables */
    strcpy (cp->name, name);
    cp->nssets = nssets;
    cp->bsize = bsize;
    cp->assoc = assoc;
    cp->policy = policy;
    cp->m = m;
    cp->hits = 0;
    cp->unique_hits = 0;
    cp->misses = 0;
    cp->writes = 0;
    cp->ignored_writes = 0;
    cp->bytes_from_j2 = 0;
    cp->bytes_from_j1 = 0;
    cp->bytes_from_j2_without_j1 = 0;
    cp->bytes_written = 0;
    cp->prefetched = 0;
cp->replacements = 0;
cp->l2_references_ltc = 0;
cp->l2_references = 0;
cp->dt_misses = 0;

cp->sets = (struct dtcache_set *) calloc (cp->nsets, sizeof (struct dtcache_set));

for (i = 0; i < nsets; i++)
    cp->sets[i].blks = (struct dtcache_blk *) calloc (cp->assoc, sizeof (struct dtcache_blk));

for (i = 0; i < nsets; i++)
    for (j = 0; j < assoc; j++) {
        cp->sets[i].blks[j].tag = 0;
        cp->sets[i].blks[j].fifo = 0;
        cp->sets[i].blks[j].status = DT_CACHE_BLK_INVALID;
        cp->sets[i].blks[j].start_address = (SS_ADDR_TYPE *) calloc (m, sizeof (SS_ADDR_TYPE));
        cp->sets[i].blks[j].end_address = (SS_ADDR_TYPE *) calloc (m, sizeof (SS_ADDR_TYPE));
    }

cp->dt_tbl = (struct dtcache_table *) calloc (cp->assoc * cp->nsets, sizeof (struct dtcache_table));
for (i = 0; i < cp->assoc * cp->nsets; i++) {
    cp->dt_tbl[i].start_addr = 0;
    cp->dt_tbl[i].end_addr = 0;
    cp->dt_tbl[i].valid = 0;
    cp->dt_tbl[i].hit = 0;
}

return (cp);

/* lookup_dcache function takes the following parameters: a pointer * to the data trace cache structure, start address of the trace to * lookup. It returns a pointer to an integer that has the number of * blocks in the trace line, the trace line is returned in dtcache_blk * structure pointer line. The function returns DT_CACHE_BLK_HIT or * DT_CACHE_BLK_MISS */

int lookup_dcache (struct dtcache *cp,
    SS_ADDR_TYPE start_addr [],
    SS_ADDR_TYPE end_addr [],
    struct dtcache_blk **line,
    unsigned int *noBB) {

    int set_num, j, found = 0, i, ctr;
    struct dtcache_blk *returnline;
    extern int l1miss;

    if (!cp) return (-1);

    set_num = 0;
    while (set_num < cp->nsets && !found) {
        j = 0;
        while (j < cp->assoc && !found) {

            return...
i = (set_num * cp->assoc) + j;

if (cp->dt_tbl[i].valid == DTCACHE_BLK_VALID &&
    address_subspace (cp->dt_tbl[i].start_addr,
    cp->dt_tbl[i].end_addr,
    start_addr [0],
    end_addr [0])) {
    ctr = 0;
    while (ctr < cp->sets [set_num].blks [i].status == DTCACHE_BLK_VALID &&
          !found) {
      if (address_subspace (cp->sets [set_num].blks [i].start_address [ctr],
                            cp->sets [set_num].blks [i].end_address [ctr],
                            start_addr [0],
                            end_addr [0])) {
        found = 1;
        *no_BB = cp->sets [set_num].blks [i].no_of_BB;
        returnline = (struct dtcache_blk *) malloc (sizeof (struct dtcache_blk));
        memcpy (returnline, &cp->sets [set_num].blks [i].sizeof (struct dtcache_blk));
        *line = returnline;
      }
      ctr++;
    }
    j++;
  }
set_num++;
}
set_num--; j--; ctr--;

/* printf ("Read [%d %d]", start_addr [0], end_addr [0]); */
if (found) {
  cp->hits++;
/* printf ("Hit [%d %d]\n", set_num, j, ctr); */
  if (cp->dt_tbl [set_num * cp->assoc + j].hit == 0) {
    cp->dt_tbl [set_num * cp->assoc + j].hit = 1;
    cp->unique_hits++;
    for (i = 0; i < cp->sets [set_num].blks [i].no_of_BB; i++)
      cp->bytes_from_dt += cp->sets [set_num].blks [i].end_address [i] -
      cp->sets [set_num].blks [i].start_address [i] + 1;
  } else {
    for (i = 0; i < cp->sets [set_num].blks [i].no_of_BB; i++)
      cp->prefetched += cp->sets [set_num].blks [i].end_address [i] -
      cp->sets [set_num].blks [i].start_address [i] + 1;
  }
return DTCACHE_HIT;
} else {
  if (cp->dt_tbl [set_num * cp->assoc + j].hit == 0)
    cp->dt_misses++;
  cp->misses++;
/* printf ("MISS\n"); */
return DTCACHE_MISS;
}

/* find_new_trace_line takes a pointer to the Data trace cache
 * structure and a new address of the trace. It finds the set number
 * and block number for the address and returns them to the calling
 * function
 */

int find_new_trace_line (struct dtcache *cp, SS_ADDR_TYPE addr) {
  int set_num, i, position;
  long long min_fifo;
  int index, min_index;

  set_num = DTCACHE_SET(cp, addr);

  for (i = 0; i < cp->assoc; i++)
    if (cp->sets[set_num].blks[i].start_address[0] == addr) return -1;

  position = 0;
  for (i = 0; i < cp->assoc; i++)
    if (cp->sets[set_num].blks[i].status == DTCACHE_BLK_INVALID) {
      cp->sets[set_num].blks[i].fifo += static_fifo;
      return i;
    }

  cp->replacements++;

  if (cp->policy == DTRandom) {
    index = (int) ( ((double) cp->assoc) * rand() / (RAND_MAX + 1.0));
    cp->sets[set_num].blks[index].fifo += static_fifo;
    return index;
  }

  if (cp->policy == DTFifo) {
    min_fifo = cp->sets[set_num].blks[0].fifo;
    min_index = 0;
    for (i = 1; i < cp->assoc; i++)
      if (cp->sets[set_num].blks[i].fifo < min_fifo) {
        min_fifo = cp->sets[set_num].blks[i].fifo;
        min_index = i;
      }
    cp->sets[set_num].blks[min_index].fifo += static_fifo;
    return (min_index);
  }

  return -1;
}

/* new_dtrace creates a new trace and adds it to the data trace cache
 * pointed to by cp. The start_address and end_address contain the
 * start and end addresses for no_BB blocks in the trace line.
 */

void new_dtrace (struct dtcache *cp,
SS_ADDR_TYPE start_addr [], SS_ADDR_TYPE end_addr [];
unsigned int no_BB { }

unsigned int set_num, index, eff_n, i;
SS_ADDR_TYPE min_start_address, max_end_address = 0;

if (cp) return;
if (no_BB > cp->m)
fatal("Error in Data Trace Cache, Basic Blocks greater than \m \%d \n", no_BB, cp->m);

eff_n = 0;
set_num = DTCACHE_SET (cp, start_addr [0]);
index = find_new_trace_line (cp, start_addr [0]);

if (index == -1) return;

for (i = 0; i < no_BB; i++) {
    cp->sets [set_num].blks [index].start_address [i] = start_addr [i];
    cp->sets [set_num].blks [index].end_address [i] = end_addr [i];
    eff_n += (end_addr [i] - start_addr [i]);
    if (eff_n > cp->bsize) {
        fprintf(stderr, "\%s Breaking out of loop because number of bytes greater than line size \%x \%x\n", cp->name, cp->bsize, eff_n);
        break;
    }
}

if (max_end_address < cp->sets [set_num].blks [index].end_address [i])
    max_end_address = cp->sets [set_num].blks [index].end_address [i];

if (min_start_address > cp->sets [set_num].blks [index].start_address [i])
    min_start_address = cp->sets [set_num].blks [index].start_address [i];
}

for (i = (set_num * cp->assoc) + index;
    cp->dt_tbl [i].start_addr = min_start_address;
    cp->dt_tbl [i].end_addr = max_end_address;
    cp->dt_tbl [i].valid = DTCACHE_BLK_VALID;
}

/* prints a trace line from the data trace cache, the parameters are
  * the set number and index */
void print_trace_line (struct dcache *cp, int set, int index) {
    int i;
    for (i = 0; i < cp->sets [set].blks [index].no_of_BB; i++)
        printf ("%d %d] start: %x, end: %x\n",
                set, index, i,
                cp->sets [set].blks [index].start_address [i],
                cp->sets [set].blks [index].end_address [i]);
    printf ("\n");
}

void scan_dtrace (struct dcache *cp, SS_ADDR_TYPE addr, int nbytes) {
    int i, j, k;
    for (i = 0; i < cp->sets; i++)
        for (j = 0; j < cp->assoc; j++)
            for (k = 0; k < cp->sets [i].blks [j].no_of_BB; k++)
            if (addr >= cp->sets [i].blks [j].start_address [k] &&
                addr <= cp->sets [i].blks [j].end_address [k])
                printf ("addr %x in set %d, block %d, entry %d\n", addr, i, j, k);

int address_subspace (SS_ADDR_TYPE b_addr,
                      SS_ADDR_TYPE a_addr,
                      SS_ADDR_TYPE s_addr,
                      SS_ADDR_TYPE e_addr) {
    if (b_addr > e_addr)
        fatal ("FATAL: address_subspace");
    if (s_addr <= b_addr && e_addr >= b_addr)
        return TRUE;
    else if (s_addr <= b_addr && e_addr >= b_addr && s_addr <= b_addr)
        return TRUE;
    else if (s_addr >= b_addr && s_addr <= b_addr && e_addr >= b_addr)
        return TRUE;
    else if (s_addr >= b_addr && s_addr <= b_addr && e_addr >= b_addr)
        return TRUE;
    return FALSE;
}

void write_invalidate (struct dcache *cp,
                       int set,
                       int index, SS_ADDR_TYPE addr,
                       unsigned int nbytes) {
    int i;
    if (cp->sets [set].blks [index].status == DTCACHE_BLK_INVALID)
        for (i = 0; i < cp->sets [set].blks [index].no_of_BB; i++)
            if (address_subspace (cp->sets [set].blks [index].start_address [i],
                                  cp->sets [set].blks [index].end_address [i],
                                  addr,
                                  addr + nbytes - 1))
                /* printf ("write update for %x to %x\n", addr, addr + nbytes - 1); */

}
/* cp->sets [set].blks [index].status = DTCACHE_BLK_INVALID; */
cp->sets [set].blks [index].fifo = ++static_fifo;
cp->writes++;
cp->bytes_written += nbytes;
cp->dtlbl [(set * cp->assoc) + index].hit = 0;
} else
 cp->ignored_writes++;

/* invalidates a trace line during write for address saddr. */

void dt_line_invalidate (struct dtcache *cp,
 SS_ADDR_TYPE saddr,
 unsigned int nbytes)
 {
 int i, set, index;
 SS_ADDR_TYPE eaddr = saddr + nbytes - 1;
 if (!cp) return;

 for (set = 0; set < cp->nsets; set++)
 for (index = 0; index < cp->assoc; index++)
 {
 i = (set * cp->assoc) + index;
 if (cp->dtlbl[i].valid == DTCACHE_BLK_VALID &&
 address_subset (cp->dtlbl[i].start_addr,
 cp->dtlbl[i].end_addr,
 saddr,
 eaddr)) {
 write_invalidate (cp, set, index, saddr, nbytes);
 } else
 cp->ignored_writes++;
 }

/* print the stats */
void dt_stats (struct dtcache* cp) {
 double a, b;

 fprintf (stderr,
 "Unique Patterns Read from Data Trace Cache : %lld \n",
 cp->unique_hits);

 fprintf (stderr,
 "L1 Hits because of DTC hits : %lld \n",
 cp->hits);

 fprintf (stderr,
 "DTC misses : %lld \n",
 cp->dt_misses);

 fprintf (stderr,
 "Total Data Trace Cache Accesses : %lld \n",
 cp->unique_hits + cp->dt_misses);

 fprintf (stderr,
 "Bytes Read from DT : %lld k\n"
cp->bytes_from_dt / 1024);

fprintf (stderr,  
"Bytes Prefetch (reuse) without incurring Miss: %lld k\n",  
cp->prefetched / 1024);

fprintf (stderr,  
"Replacements in Data Trace Cache : %lld \n",  
cp->replacements);

fprintf (stderr,  
"Write Updates to Data Trace Cache : %lld \n",  
cp->writes);

fprintf (stderr,  
"Bytes Written to Data Trace Cache : %lld k\n",  
cp->bytes_written / 1024);

fprintf (stderr,  
"Bytes Read from L2 with Data Trace Cache : %lld k\n",  
cp->bytes_from_l2 / 1024);

fprintf (stderr,  
"Bytes Read from L2 without Data Trace Cache : %lld k\n",  
cp->bytes_from_l2_without_dt / 1024);

fprintf (stderr,  
"L2 accesses without Data Trace Cache : %lld\n",  
cp->l2_references);

fprintf (stderr,  
"L2 accesses with Data Trace Cache : %lld\n",  
cp->l2_references_dtc);

a = (cp->bytes_from_l2 + cp->bytes_from_dt);  
b = (cp->bytes_from_l2_without_dt + cp->prefetched);

fprintf (stderr,  
"Efficiency: %g\n\n", ((b - a)/b) * 100);

fprintf (stderr,  
"%n**** %d k , %dL , %ds , %da ****\n",  
dtmsize * dtset * dtassoc, dtmsize, dtset, dtassoc);  
}
A.3  dtcache_params.h – Data trace cache parameter

#ifndef CACHE_PARAMS
#define CACHE_PARAMS

#define dtsize 128  /* size of the line */
#define dtset  8    /* number of sets */
#define dtassoc 4   /* associativity */
#define dtbsize sizeof (char)  /* data/address block size */
#define FALSE 0      /* max. number of basic blocks */
#define TRUE 1

#define DTCACHE1 1    /* Data trace cache model 1 */
#define DTCACHE2 0    /* Data trace cache model 2 */
#define DTCACHE3 0    /* Data trace cache model 3 */

#endif
A.4  dt_l1_link.c – Links sim-cache and Data trace cache

#include "dtcache.h"
#include "dtcache_params.h"

extern struct dtcache *dtcache1, *dtcache2, *dtcache3;
extern struct dtcache_blk dtcache1blk, dtcache2blk;

static int no_bytes1, no_bytes2, no_bytes3;
extern int l1miss, first_l1miss;
static int mvalue1, mvalue2, mvalue3;
static SS_ADDR_TYPE start_addr1[dtmsize], end_addr1[dtmsize],
    start_addr2[dtmsize], end_addr2[dtmsize], tempaddr[1],
    start_addr3[dtmsize], end_addr3[dtmsize], globaladdr;

/* function for debugging, calculate size of trace line getting added */
void
calcsize(SS_ADDR_TYPE tstart_addr[], SS_ADDR_TYPE tend_addr[], int value)
{
    int i = 0;
    int eff_n = 0;
    for (i = 0; i < value; i++) {
        eff_n += (tend_addr[i] - tstart_addr[i]) + 1;
        if (eff_n > dtmsize) {
            printf("ERR = %d\n ", eff_n);
            break;
        }
    }
}
/* function for merging the blocks already found in the trace line
* contiguous blocks into the same block and compacting the trace line */
void
mergestart(unsigned int mvalue, SS_ADDR_TYPE *start_addr,
            SS_ADDR_TYPE *end_addr, unsigned int *no_bytes, int index,
            unsigned int nbytes)
{
    int i = 0;

    *no_bytes = *no_bytes - (*end_addr[index] - start_addr[index] + 1);

    if ((*no_bytes + nbytes) > dtmsize) {
        end_addr[index] = start_addr[index] + dtmsize - (*no_bytes) - 1;
        *no_bytes = dtmsize;
    } else {
        *no_bytes += nbytes;
        end_addr[index] = start_addr[index] + nbytes - 1;
    }

    for (i = 0; i < mvalue; i++) {
        if (end_addr[index] == start_addr[index]) {
            // code...
/* This function merges the new block created, if it has contiguous data
 * some of the already existing blocks of the trace line and compacts the
 * trace line, this function returns 1 if there has been a merging or
 * compaction, else it returns a 0 */

int mergeendstart(unsigned int mvalue, SS_ADDR_TYPE * start_addr,
                  SS_ADDR_TYPE * end_addr, unsigned int *no_bytes)
{
    int i = 0;
    for (i=0;i<mvalue;i++) {
        if (start_addr[mvalue] == end_addr[i] + 1 || start_addr[mvalue] == end_addr[i]) {
            *no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] + 1);
            end_addr[mvalue] = end_addr[i];
            *no_bytes = *no_bytes + (end_addr[i] - start_addr[i] + 1);
            return 1;
        }
    }
    if (end_addr[mvalue] + 1 == start_addr[i] || end_addr[mvalue] == start_addr[i]) {
        *no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] + 1);
        start_addr[i] = start_addr[mvalue];
        *no_bytes = *no_bytes + (end_addr[i] - start_addr[i] + 1);
        return 1;
    }
    if (start_addr[mvalue] < start_addr[i] && end_addr[mvalue] > end_addr[i]) {
        *no_bytes = *no_bytes - (end_addr[i] - start_addr[i] + 1);
        start_addr[i] = start_addr[mvalue];
        end_addr[i] = end_addr[mvalue];
        return 1;
    }
    if (end_addr[mvalue] == end_addr[i] && start_addr[mvalue] >= start_addr[i]) {
        *no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] + 1);
        return 1;
    }
    if (end_addr[mvalue] == end_addr[i] && start_addr[mvalue] < start_addr[i]) {
        *no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] + 1);
        start_addr[i] = start_addr[mvalue];
        *no_bytes = *no_bytes - start_addr[i] + end_addr[i] + 1;
        return 1;
    }
    if (end_addr[mvalue] > end_addr[i] && start_addr[mvalue] > start_addr[i] && start_addr[mvalue] < end_addr[i]) {
        *no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] + 1);
        *no_bytes = *no_bytes - (end_addr[i] - start_addr[i] + 1);
    }
DATA TRACE CACHE SOURCE CODE

```c
end_addr[i] = end_addr[mvalue] ;
*no_bytes = *no_bytes + (end_addr[i] - start_addr[i] +1);
return 1;
}
if (end_addr[mvalue]>start_addr[i] && start_addr[mvalue]<start_addr[i] &&
end_addr[mvalue]<end_addr[i]) {
*no_bytes = *no_bytes - (end_addr[mvalue] - start_addr[mvalue] +1);
*no_bytes = *no_bytes - (end_addr[i] - start_addr[i] +1);
start_addr[i] = start_addr[mvalue];
*no_bytes = *no_bytes + (end_addr[i] - start_addr[i] +1);
return 1;
}
return 0;

/* function to increment the bytes of the current trace line while checking for */
/* the maximum line size and also the maximum number of non-contiguous blocks */
/* that can be present per trace line */

void
incrementbytes(unsigned int *no_bytes, SS_ADDR_TYPE *start_addr,
                 SS_ADDR_TYPE *end_addr, unsigned int mvalue,
                 unsigned int nbytes)
{
    if (((*no_bytes) + nbytes) > dtsize) {
        end_addr[mvalue] = start_addr[mvalue] + dtsize - (*no_bytes) -1;
        *no_bytes = dtsize;
    } else {
        (*no_bytes) += nbytes;
        end_addr[mvalue] = start_addr[mvalue] + nbytes -1;
    }
}

/* This function does the actual building of the trace line, */
/* maintains the line in a buffer before transferring the line */
/* to the Data Trace Cache */

void push_addr(SS_ADDR_TYPE baddr, unsigned int nbytes)
{
    unsigned int i, j, foundindex1, foundindex2, foundindex3;
    int found1 = FALSE;
    int found2 = FALSE;
    int found3 = FALSE;

    i = j = foundindex1 = foundindex2 = foundindex3 = 0;
```
for (i = 0; i < mvalue1; i++) {
    if (baddr == start_addr1[i]) {
        found1 = TRUE;
        foundindex1 = i;
    }
}

for (i = 0; i < mvalue2; i++) {
    if (baddr == start_addr2[i]) {
        found2 = TRUE;
        foundindex2 = i;
    }
}

for (i = 0; i < mvalue3; i++) {
    if (baddr == start_addr3[i]) {
        found3 = TRUE;
        foundindex3 = i;
    }
}

if (found1 == FALSE && found2 == FALSE && found3 == FALSE) {

    if (DTCACHE1) {
        if (llmiss == TRUE) {
            start_addr1[mvalue1] = baddr;
            incrementbytes(&no_bytes1, start_addr1, end_addr1, mvalue1, nbytes);

            if (mergeoendstart(mvalue1, start_addr1, end_addr1, &no_bytes1) == 1)
                mvalue1--;

            if (no_bytes1 >= dts) || mvalue1 + 1 >= dts) {
                new_dtrace(dtcache1, start_addr1, end_addr1, mvalue1 + 1);
                mvalue1 = 0;
                no_bytes1 = 0;
            } else
                mvalue1++;
        }
    }

    if (DTCACHE2) {
        start_addr2[mvalue2] = baddr;
        incrementbytes(&no_bytes2, start_addr2, end_addr2, mvalue2, nbytes);

        if (mergeoendstart(mvalue2, start_addr2, end_addr2, &no_bytes2) == 1)
            mvalue2--;

        if (no_bytes2 >= dts || mvalue2 + 1 >= dts) {
            new_dtrace(dtcache2, start_addr2, end_addr2, mvalue2 + 1);
            mvalue2 = 0;
            no_bytes2 = 0;
        }
    }
}
else
    mvalue2++;  
}

if (DTCACHE3) {
    if (first_lmiss == TRUE) {
        start_addr3[mvalue3] = badhr;
        incrementbytes(&no_bytes3, start_addr3, end_addr3, mvalue3, nbytes);

        if (mergeendstart(mvalue3, start_addr3, end_addr3, &no_bytes3) == 1)
            mvalue3--;  
        else {
            mvalue3++;  
        }
    }

    else {
        if (found1 && DTCACHE1) {
            if (lmiss == TRUE) {
                mvalue1--;  
                mergestart(mvalue1, start_addr1, end_addr1, &no_bytes1, foundindex1, nbytes);

                if (no_bytes1 >= dtsize || mvalue1 + 1 >= dtsize) {
                    new_dttrace(dtcache1, start_addr1, end_addr1, mvalue1 + 1);
                    mvalue1 = 0;
                    no_bytes1 = 0;
                }

                else
                    mvalue1++;  
            }

        }

        if (found2 && DTCACHE2) {
            mvalue2--;  
            mergestart(mvalue2, start_addr2, end_addr2, &no_bytes2, foundindex2, nbytes);

            if (no_bytes2 >= dtsize || mvalue2 + 1 >= dtsize) {
                new_dttrace(dtcache2, start_addr2, end_addr2, mvalue2 + 1);
                mvalue2 = 0;
                no_bytes2 = 0;
            }

        }

    }

}
} else
    mvalue2++;  
}

if (found3 & DTCACHE3) {
    if (firstLMiss == TRUE) {
        mvalue3--;  
        mergestart(mvalue3, start_addr3, end_addr3, &no_bytes3,  
                  foundindex3, nbytes);

        if (no_bytes3 >= dtsize || mvalue3 + 1 >= dtsize) {
            new_dtttrace(dttcache3, start_addr3, end_addr3,  
                         mvalue3 + 1);
            mvalue3 = 0;
            no_bytes3 = 0;
            firstLMiss = FALSE;
        } else
            mvalue3++;  
    }
}

LMiss = FALSE;
}
B Detailed Simulation Data

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8K L1 4K DTC (1 Way)</th>
<th>8K L1 4K DTC (2 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>8K L1 8K DTC (4 Way)</th>
<th>8K L1 4K DTC (64lines)</th>
<th>16K L1 4K DTC (4 Way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bh</td>
<td>380820</td>
<td>280821</td>
<td>266367</td>
<td>274269</td>
<td>202974</td>
<td>194676</td>
</tr>
<tr>
<td>Bsort</td>
<td>235878</td>
<td>119606</td>
<td>118609</td>
<td>118888</td>
<td>105128</td>
<td>119075</td>
</tr>
<tr>
<td>Ft</td>
<td>1283975</td>
<td>802340</td>
<td>912280</td>
<td>941743</td>
<td>318587</td>
<td>552866</td>
</tr>
<tr>
<td>Health</td>
<td>253533</td>
<td>203226</td>
<td>203935</td>
<td>207218</td>
<td>174196</td>
<td>187536</td>
</tr>
<tr>
<td>Ks</td>
<td>1039304</td>
<td>260702</td>
<td>209064</td>
<td>189257</td>
<td>139944</td>
<td>99948</td>
</tr>
<tr>
<td>Mst</td>
<td>223508</td>
<td>196853</td>
<td>200300</td>
<td>202386</td>
<td>189640</td>
<td>193350</td>
</tr>
<tr>
<td>Perimeter</td>
<td>570193</td>
<td>282901</td>
<td>282773</td>
<td>286896</td>
<td>279986</td>
<td>282226</td>
</tr>
<tr>
<td>Simple</td>
<td>123766</td>
<td>20409</td>
<td>17920</td>
<td>16761</td>
<td>15984</td>
<td>15484</td>
</tr>
<tr>
<td>Treaddd</td>
<td>184509</td>
<td>152233</td>
<td>150753</td>
<td>151472</td>
<td>150279</td>
<td>150292</td>
</tr>
<tr>
<td>Tsp</td>
<td>307421</td>
<td>76952</td>
<td>119241</td>
<td>153623</td>
<td>96424</td>
<td>106178</td>
</tr>
<tr>
<td>Voronoi</td>
<td>73386</td>
<td>48005</td>
<td>47478</td>
<td>47899</td>
<td>44942</td>
<td>73386</td>
</tr>
</tbody>
</table>

Table 6: L2 References With Data Trace Cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8K L1 4K DTC (1 Way)</th>
<th>8K L1 4K DTC (2 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>8K L1 8K DTC (4 Way)</th>
<th>8K L1 4K DTC (64lines)</th>
<th>16K L1 4K DTC (4 Way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bh</td>
<td>0.841</td>
<td>0.786</td>
<td>0.805</td>
<td>0.605</td>
<td>0.677</td>
<td>0.870</td>
</tr>
<tr>
<td>Bsort</td>
<td>0.705</td>
<td>0.723</td>
<td>0.619</td>
<td>0.664</td>
<td>0.627</td>
<td>0.705</td>
</tr>
<tr>
<td>Ft</td>
<td>0.633</td>
<td>0.696</td>
<td>0.717</td>
<td>0.503</td>
<td>0.653</td>
<td>0.839</td>
</tr>
<tr>
<td>Health</td>
<td>0.899</td>
<td>0.892</td>
<td>0.892</td>
<td>0.820</td>
<td>0.846</td>
<td>0.908</td>
</tr>
<tr>
<td>Ks</td>
<td>0.905</td>
<td>0.912</td>
<td>0.934</td>
<td>0.871</td>
<td>0.829</td>
<td>0.511</td>
</tr>
<tr>
<td>Mst</td>
<td>0.948</td>
<td>0.932</td>
<td>0.925</td>
<td>0.885</td>
<td>0.897</td>
<td>0.934</td>
</tr>
<tr>
<td>Perimeter</td>
<td>0.536</td>
<td>0.567</td>
<td>0.567</td>
<td>0.557</td>
<td>0.553</td>
<td>0.879</td>
</tr>
<tr>
<td>Simple</td>
<td>0.182</td>
<td>0.201</td>
<td>0.152</td>
<td>0.145</td>
<td>0.180</td>
<td>0.557</td>
</tr>
<tr>
<td>Treaddd</td>
<td>0.848</td>
<td>0.838</td>
<td>0.842</td>
<td>0.838</td>
<td>0.836</td>
<td>0.922</td>
</tr>
<tr>
<td>Tsp</td>
<td>0.547</td>
<td>0.672</td>
<td>0.679</td>
<td>0.611</td>
<td>0.501</td>
<td>0.906</td>
</tr>
<tr>
<td>Voronoi</td>
<td>0.716</td>
<td>0.723</td>
<td>0.736</td>
<td>0.699</td>
<td>0.693</td>
<td>0.833</td>
</tr>
</tbody>
</table>

Table 7: Data Trace Cache Miss Rates
### Table 8: L1 Miss Rate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8K L1 DTC (1 Way)</th>
<th>8K L1 DTC (2 Way)</th>
<th>8K L1 DTC (4 Way)</th>
<th>8K L1 DTC (64 lines)</th>
<th>16K L1 DTC (4 Way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bh</td>
<td>0.0160</td>
<td>0.0118</td>
<td>0.0112</td>
<td>0.0115</td>
<td>0.0085</td>
</tr>
<tr>
<td>Bsort</td>
<td>0.0141</td>
<td>0.0072</td>
<td>0.0071</td>
<td>0.0071</td>
<td>0.0063</td>
</tr>
<tr>
<td>Ft</td>
<td>0.4641</td>
<td>0.2900</td>
<td>0.3298</td>
<td>0.3404</td>
<td>0.1152</td>
</tr>
<tr>
<td>Health</td>
<td>0.1463</td>
<td>0.1172</td>
<td>0.1177</td>
<td>0.1196</td>
<td>0.1006</td>
</tr>
<tr>
<td>Ks</td>
<td>0.1912</td>
<td>0.0480</td>
<td>0.0385</td>
<td>0.0348</td>
<td>0.0257</td>
</tr>
<tr>
<td>Mst</td>
<td>0.0787</td>
<td>0.0693</td>
<td>0.0705</td>
<td>0.0712</td>
<td>0.0668</td>
</tr>
<tr>
<td>Perimeter</td>
<td>0.0375</td>
<td>0.0186</td>
<td>0.0186</td>
<td>0.0189</td>
<td>0.0184</td>
</tr>
<tr>
<td>Simple</td>
<td>0.0095</td>
<td>0.0016</td>
<td>0.0014</td>
<td>0.0013</td>
<td>0.0012</td>
</tr>
<tr>
<td>Treesadd</td>
<td>0.0182</td>
<td>0.0150</td>
<td>0.0149</td>
<td>0.0150</td>
<td>0.0149</td>
</tr>
<tr>
<td>Tsp</td>
<td>0.0845</td>
<td>0.0211</td>
<td>0.0328</td>
<td>0.0422</td>
<td>0.0265</td>
</tr>
<tr>
<td>Voronoi</td>
<td>0.0240</td>
<td>0.0157</td>
<td>0.0155</td>
<td>0.0157</td>
<td>0.0147</td>
</tr>
</tbody>
</table>

### Table 9: Unique Patterns in the Data trace Cache
Table 10: Replacement Rate of Data Trace Cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8K L1 4K DTC (1 Way)</th>
<th>8K L1 4K DTC (2 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>8K L1 8K DTC (4 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>16K L1 4K DTC (4 Way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bh</td>
<td>0.069 0.057 0.048</td>
<td>0.063 0.059 0.053</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsort</td>
<td>0.057 0.045 0.036</td>
<td>0.044 0.050 0.040</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ft</td>
<td>0.727 0.558 0.508</td>
<td>0.585 0.595 0.442</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health</td>
<td>0.061 0.055 0.053</td>
<td>0.056 0.068 0.053</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ks</td>
<td>0.165 0.123 0.070</td>
<td>0.114 0.345 0.074</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mst</td>
<td>0.066 0.050 0.049</td>
<td>0.052 0.064 0.049</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perimeter</td>
<td>0.041 0.037 0.036</td>
<td>0.035 0.046 0.049</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>0.005 0.006 0.004</td>
<td>0.003 0.004 0.026</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treeadd</td>
<td>0.044 0.043 0.043</td>
<td>0.043 0.053 0.046</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsp</td>
<td>0.264 0.100 0.073</td>
<td>0.098 0.152 0.046</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voronoi</td>
<td>0.047 0.042 0.039</td>
<td>0.039 0.052 0.045</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Improvement in L1 Hit Rate because of Data Trace Cache Hits

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>8K L1 4K DTC (1 Way)</th>
<th>8K L1 4K DTC (2 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>8K L1 8K DTC (4 Way)</th>
<th>8K L1 4K DTC (4 Way)</th>
<th>16K L1 4K DTC (4 Way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bh</td>
<td>0.405 0.401 0.317</td>
<td>0.780 0.654 0.245</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bsort</td>
<td>0.939 0.775 0.621</td>
<td>0.866 0.752 0.574</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ft</td>
<td>5.655 3.340 2.794</td>
<td>9.507 5.617 1.782</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health</td>
<td>0.247 0.222 0.202</td>
<td>0.374 0.299 0.181</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mst</td>
<td>0.165 0.108 0.097</td>
<td>0.165 0.144 0.081</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perimeter</td>
<td>0.671 0.606 0.586</td>
<td>0.607 0.615 0.261</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>0.925 1.185 0.967</td>
<td>0.977 1.257 0.538</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treeadd</td>
<td>0.180 0.188 0.184</td>
<td>0.191 0.190 0.092</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsp</td>
<td>4.254 1.321 0.802</td>
<td>1.524 1.799 0.155</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voronoi</td>
<td>0.455 0.418 0.393</td>
<td>0.444 0.453 0.244</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>