DATA TRACE CACHES - CAN THEY WORK ?

Suresh Warrier

Department of Computer Science
University of Texas at Austin
sew@cs.utexas.edu

Abstract

Transistor density has been increasing 50% per year which along with increases in die sizes are leading to 60% to 80% growth per year in transistor count on a chip. The day is not far off when we will have a billion transistors available on a single chip. This presents opportunities for new levels of computing capability. In this report, I introduce the concept of data trace caches, an idea of Dr. Doug Burger of the Computer Science Department, University of Texas at Austin. A data trace cache contains previously used global data packed in the order that they were accessed. I demonstrate how this is a promising concept which needs to be studied further.

1. Introduction

The Instruction Trace Cache has become an integral part of modern ILP processor architecture discussion. A large sophisticated trace cache provides a logically contiguous instruction stream unlike the physical instruction stream which is not contiguous[1]. These trace cache lines pack instructions that are accessed together. Research shows that the instruction trace cache is effective in delivering more than one basic block per cycle and continues to add performance as the size of the storage structure is increased. The trace cache is definitely going to play a big role with billion-transistor chips. It reduces latency delays caused because of instruction cache misses.

There have have also been numerous papers on efforts to reduce data cache misses.[2,3,4,5,6,7]. They have either involved using various speculation techniques as in [2,3,4], and/or involved prefetching of data as in [5,6,7]. These papers demonstrate that most data cache misses are related to accessing linked data structures.

This report introduces the *A data trace cache* which is similar to the instruction trace cache in that it packs contiguously accessed data much like the instruction trace cache packs contiguously executed instructions. The data trace cache is a very general idea and has real benefits in the billion-transistor chips of the future. However, in this report, we limit ourselves to studying the effect of using data trace caches in reducing data cache misses and for that purpose we concentrate on linked data structures. It has been shown that pointer loads represent a large fraction of all loads in the Olden benchmarks and contribute a disproportionately larger fraction of the cache misses, accounting for nearly all the misses for many of the programs [5].

The rest of this report is organized as follows. We begin with a discussion of data trace caches and how it can be used in billion-transistor chips in section 2. In section 3, we explain what the motivation for the project and what we sought to accomplish. A description of our mechanism is presented in section 4, followed by a quantitative evaluation of the results in section 5. We present our conclusions and discuss future possibilities in section 6.

2. Data Trace caches

A data trace cache is a concept very similar to an instruction trace cache. That being said, this is an idea in its infancy and what exactly should constitute a data trace cache is not very well-known at this stage. We could assume that the data trace consists of numerous cache lines, with each cache line consisting of cache blocks containing data that were accessed contiguously in some computation window.

The concept of the data trace becomes very powerful when considering multiple CPUs on the billion-transistor chips of tomorrow. If all the data associated with a computation can be collected in a cache line, then that cache line can be transported to a free CPU to start execution. In fact, we could visualize a CPU dedicated to collecting cache lines of data associated with computations for different processes, and scanning for free CPUs to which the process context along with associated data cache lines are transported so that the CPU could execute that piece of computation. The data trace cache lines would be independent of regular data caches and could even allow CPUs to execute when stalled because of cache misses.

Of course, these are just ideas and before anything practical can come from them, we need to understand the dynamics of data accesses especially data accessed during computations.

Unlike instructions where the property of contiguity is easily identified, the same cannot be said for data. Instead, we define contiguous data as data accessed in successive load instructions. Data addresses of loads from stack are not predictable and can change for every pass through a function. There is not much gain in saving such addresses. Also, stack loads are private for each CPU and cannot be globally transported. So, we will concentrate only on loads of heap data.

3. Motivation

Linked data structures are widely used in compilers, databases and graphics applications. They are constructed by connecting data elements to one another explicitly; the structure contains fields that name all adjacent elements by address. The structures are often allocated at random times during the execution of the program when the need for an additional structure is discovered which causes their locations to be spread all over memory. This dynamic construction allows the linked data structure to grow very large and also difficult to cache. Added to this is the fact that accesses to successive elements of the list and to the data they contain cannot be overlapped because the process of address generation itself requires an inherently serial evaluation through memory. This is commonly known as the pointer-chasing problem and effectively exposes the full latency of each access.

Most efforts at hiding this latency have involved early prefetching of the data structures while overlapping them with other work. Both software-based[6] and hardware-based[5] as well as a combination of both approaches have been proposed.[7].

Figure 1 shows an example of the pointer loads and a linked data structure traversal program.

```
for (insn = f; insn; insn = insn->next)
        process(insn->pat->code);
```

C - program fragment

```
        1. lw $24, f            ; insn = f
        2. beq $24, $0, $Exit
$Head: 3. lw $15, 8($24)        ; insn->pat
        4. lw $4, 0($15)        ; insn->pat->code
        5. jal process
        6. lw $24, 4($24)       ; insn = insn->next
        7. bne $24, $0, $Head
$Exit:
```

Assembler fragment

Instruction 6 = recurrent load
Instruction 3 = traversal load
Instruction 4 = data load

Fig. 1

In this report, we use the pointer-chasing problem as a background to study the impact of data trace caches, although as mentioned before the concept of data trace is a more general one. With data trace caches, there is no prefetching involved. It is assumed that the data can be placed in the data trace cache simultaneously with the data cache. However, not all memory accesses cause data to be put in the data trace cache. This report presents a methodology to choose which data accesses are ideally suitable for placement in the data trace.

Designing a data trace cache requires investigating many issues like:

- Construction (based on dependences)
- Reconstruction (when pointers change)
- Stores (finding, lookup)
- Coherence

However, our report is motivated by the need to provide answers to some important questions before we can even begin to design a real data trace cache. This report thus proposes to answer the following questions:

1. Is there value in packing associated data together ?
2. How often are these data accessed together within a certain instruction cycle window ?
3. How often is other data accessed during this computation window ?

The success of a data trace cache is based on answers to these questions and these provide the premises which determine if this whole idea is practical.

As part of this investigation we also propose one way to build a data trace cache.

4. Challenges and design

There were numerous challenges to overcome especially as there was no related work to draw inspiration from. However, the work done in [5] demonstrated how a linked data structure could be represented and a similar methodology could be employed for our purposes.

The chief challenges involved collecting the dependence load instances and determining how often they were accessed together. A data trace cache must be constructed and the depenendent load instances had to be linked to a trace cache line. Information about how often the addresses were being accessed had to be collected. A big problem was not knowing what format a data trace cache would look like, would it consist of data elements, would it consist of data blocks ?

Ignoring stack loads was not really a problem because stack addresses conventionally lie within a certain address range. Ignoring spurious loads, however, was a bigger challenge. These are rarely executed loads or loads of data, and not loads of addresses.

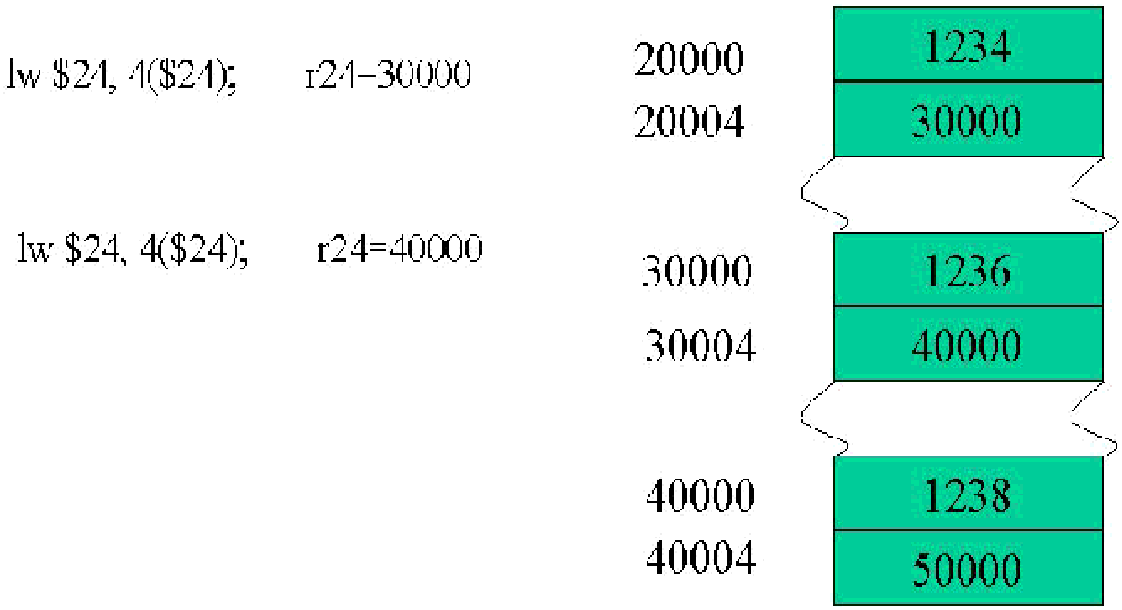
4.1 Collecting the data dependences

In this section we describe how we collect the data dependences and how we build a data trace cache table. The methodology involves constructing two tables, which I call the *pointer load table* and the *data trace table*.

In this section we explain how we intend to capture dependent loads. The basic assumption is that numeric values associated with addresses are rarely seen in other contexts and therefore two loads which name the same memory address are related. The additional requirement is that the first load actually loads this address, and the second load uses the same address as a base to load another value from memory. Fig 2 explains this procedure with an example.

Our methodology involves constructing two new tables, which I call the i>pointer load table and the *data trace table*.

Fig 3 shows the block schematic diagram with the pointer load table and the data trace table.



Assume r24 had a value of 20000 when executing line 6 of the assembler fragment of fig1. After the first execution above, r24 will equal 30000. This is a load instruction which loads a value equal to 30000. On the next iteration through the loop, the load uses this base address of 30000 to do another load and load a value of 40000. The addresses 30000 and 40000 are related and constitute a pair of dependent load addresses.

Fig. 2

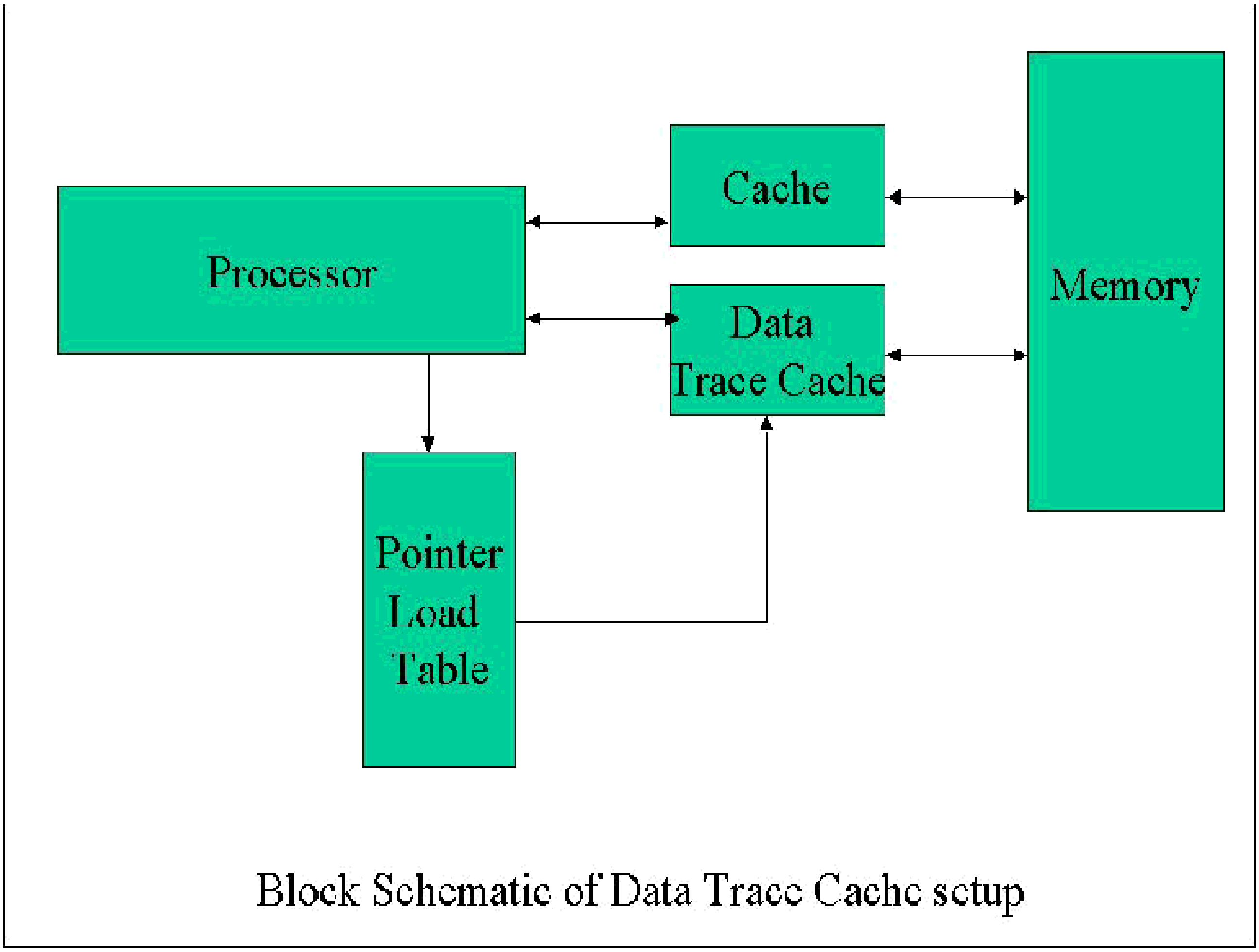


Fig 3

4.2 Pointer Load table

The pointer load table is used to save all pointer loads. In reality since this cannot be determined easily all load instructions would be saved in the table. The memory addresses being loaded and the instruction cycle in which the load was executed is saved. When the same memory address is

The pointer load table can be implemented as a cyclical queue or a cache containing the load addresses and cycle value pairs. They can be hashed or indexed by the address values for easy access. The table can never overflow as it wraps within the cache always replacing the oldest address.

A future design must optimize the table, probably with some help from the compiler to only record real pointer loads, not data loads, like line 4 of the assembler fragment in figure 1.

4.3 Data Trace Cache Table

I have conveniently called the second table a data trace cache table, although this is not intended to be a proposal for the structure of a future data trace cache.

The data trace cache table is where the contiguous sequence of dependence loads are captured. Each sequence of contiguous loads are recorded in a separate entry in the table which is called a data trace cache line. For now, we do not restrict the number of elements in a cache line. A true data cache line could not allow unlimited number of entries.

A new data trace cache line is created whenever a second load is executed whose base address is already available in the pointer load table. If the base address is already present in the a cache line, then the address gets added to the cache line. Whenever a cache line is created or a new address is added to an existing cache line, the total access cycle which represents the length of the computation window where the addresses are being accessed must be calculated. This is a crucial element of the design as this enables to determine how often the addresses are being accessed (i.e., loads using these addresses occur) in a computation window, how many external addresses are being invoked, and important characteristics. We recalculate this computation window length when we add a new element because we make the reasonable assumption that linked lists with more elements require a longer cycle to complete their processing.

For now we have ignored deletions from the list. We make an assumption that most freeing of linked lists happen at the end of the program and do not cause significant impact.

The big challenge is here is handling spurious, non-recurrent (one time-only) loads as these can reduce the effective size of both the pointer load and data trace cache tables.

4.4 Data collected

With the two tables constructed above, we are now in a position to set up and calculate some important information for evaluation of data trace cache effectiveness.

The following are some important statistics that can be gathered:

1. How often data in cache line accessed together in one cycle?
2. How many distinct cache blocks ?
3. Size of trace lines.
4. Length of access cycle time.

5. Experimental Framework

In this section, we describe how we implemented our design, what trade-offs we faced and the results we obtained.

We chose the Simple Scalar toolset as the simulation tool for our experiments as it already had a dispatch cycle block which could be modified to construct and maintain our tables. We added the pointer load and data trace tables as arrays of structures. The pointer load was a simple array. However, the data trace table was an array of cache line headers which was the head for a linked list of cache elements. To facilitate address matching, we also employed a simple hashing algorithm to index into the array elements in both the tables. We had a 128-entry pointer load table and a 128-entry data cache table. Within a cache table entry, we allowed unlimited cache elements in a linked list.

We modified the out-of-order simulator to call our routine whenever a non-stack load was discovered and updated the pointer load table and data trace table entries. We added a number of counters in the cache header structure to keep running totals of information we were interested in.

We ran some of the Spec95 benchmarks on our modified simulator using the default values for the L1, L2 and TLB caches. Our platform was an IBM RS/6000 machine running AIX 4.3.0. These experiments should be conducted with the Olden benchmarks to get more conclusive results. For our tests, we chose the integer benchmarks *gcc(cc1)* and *li*; and the floating point benchmarks *fpppp*, *su2cor* and *hydro2*.

We discovered that for some of the larger benchmarks that we quickly ran out of entries in the data trace table. We employed some sophisticated algorithms to determine which cache lines seemed least promising based of number of elements, no of times the cache line had already been accessed, last accessed cycle, etc.

Figures 4, 5, 6 show some of the interesting data that we collected.

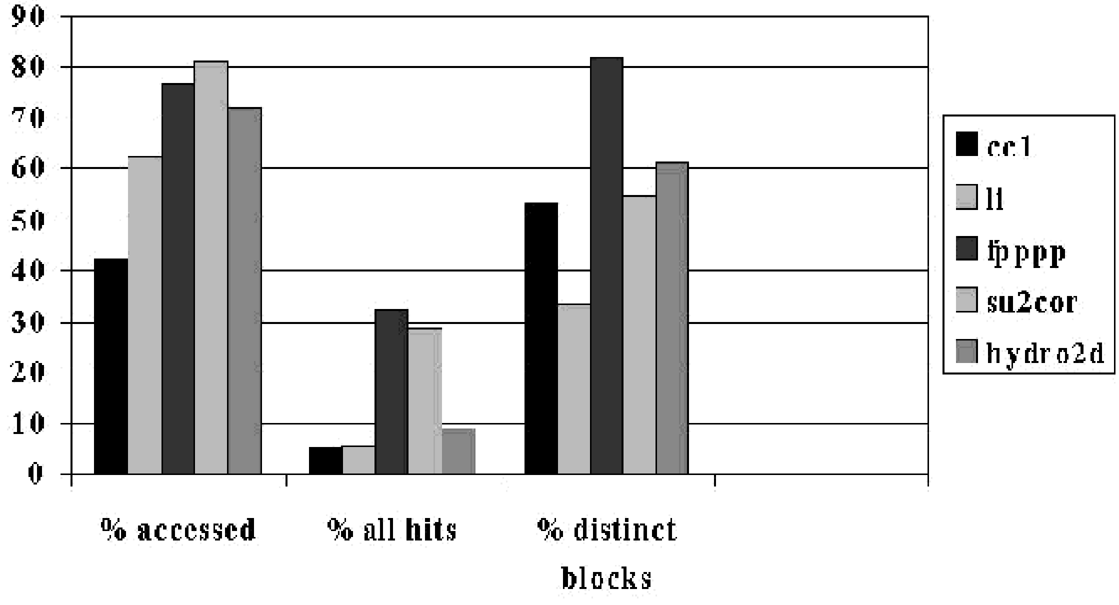


Fig 4

This graph plots a few statistics collected for each benchmark which prove that the data trace cache is a useful concept. The **%accessed** refers to the percentage of elements accessed during an active computation cycle for that trace cache line. The **% all hits%** refers to the loads during a computation window that only included elements from that particular cache line. The **%distinct blocks** refers to the number of distinct cache blocks that the addresses in a cache line belonged to. All values are averages across the all valid trace lines - trace lines which were accessed only once or had only 2 elements were not considered - across the computation window for that trace line.

Based on these, we observe that a large percentage of elements collected in a cache line get accessed together more than 50% of the time except for *gcc*. This anomaly can be explained because *gcc* had large trace cache lines and if we had restricted number of elements in the cache line might have performed better. Our assumptions and design to collect dependence loads seems to be substantiated because we have been able to collect data addresses that get accessed often in a computation window.

Another interesting, although surprising observation for me was the huge number of non-trace lines accesses during a computation window. I am at a loss to explain this and need to perhaps conduct some more experiments to clarify this. Even taking into account, accesses in other cache lines (the figures are not shown on the graph, but the average improvement is only about 30-40%) this seems pretty big. This would defeat the idea of associating cache lines of data with a computation. Of course detailed study by varying the computation window size, checking for accesses other cache lines, increasing cache block size, etc. may be need to be undertaken before making any definite conclusions.

Finally, as befitting linked data structures whose loads we have tried to capture, the number of distinct blocks for the addresses belonging to a cache line are generally more than 50% except for *li*.

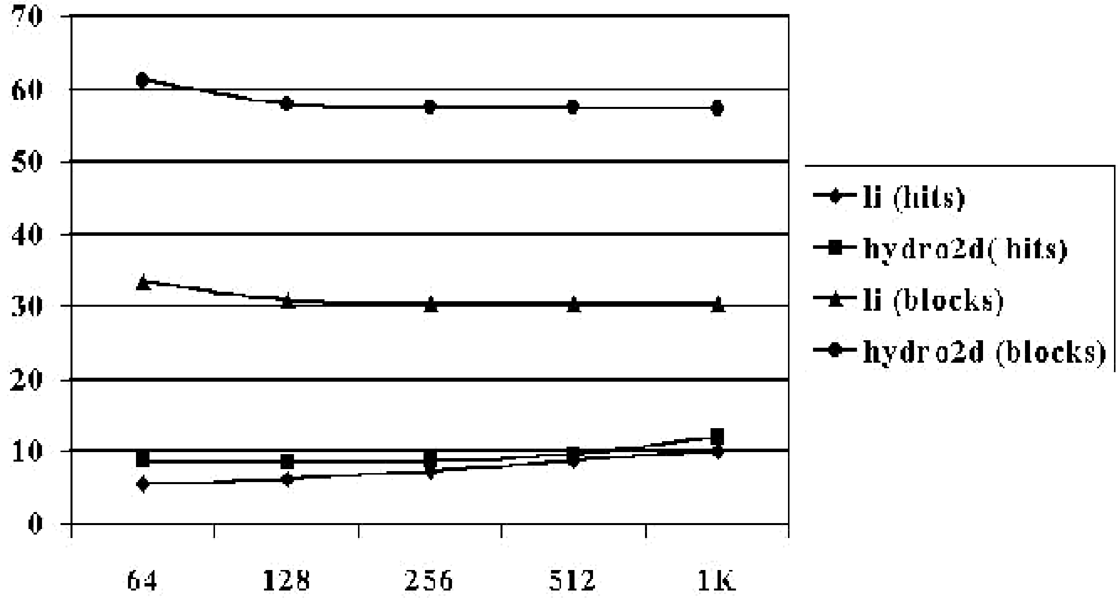


Fig 5

Figure 5 plots percentage of loads that only used addresses within a data trace cache line during a computation window and the percentage of distinct blocks within a cache line versus size of the trace cache block size. It appears that increasing the cache block size slowly decreases the number of distinct blocks and slowly improves the hit percentage. It does not appear that the solution to improving hits lies with increasing cache block sizes.

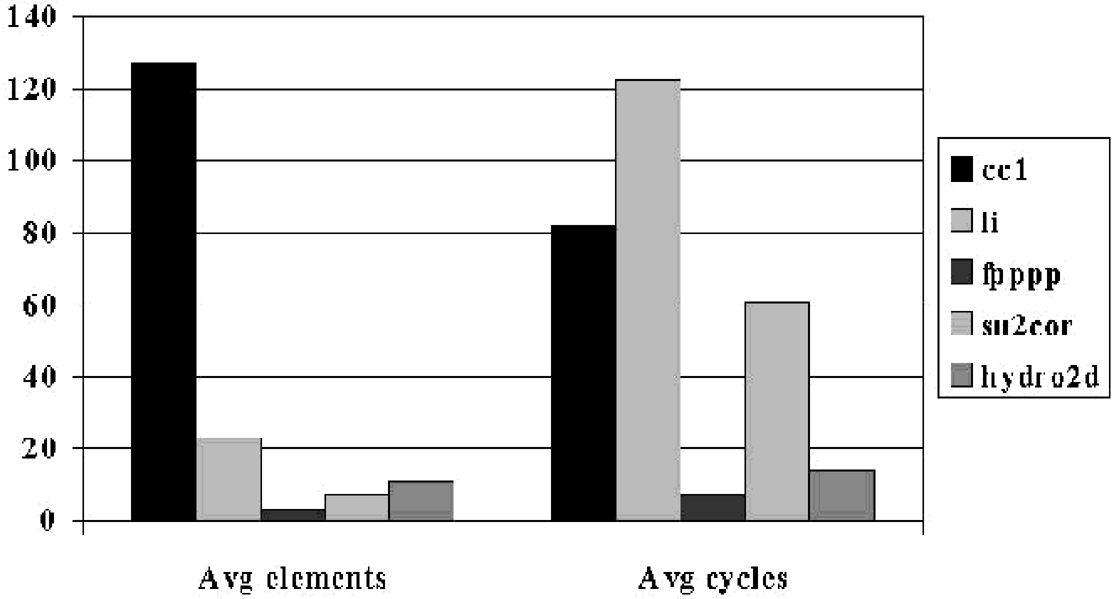


Fig 6

Figure 6 just compares the average number of times a computation window is executed and the average number of elements in a trace line. The integer benchmarks have large number of elements in a trace cache line and they also show more regularity in executing with the same data. The floating point benchmarks have smaller trace cache lines and generally execute the same computation window less often.

These figures are biased in the sense they only collect data for computation windows that access linked data structures. As expected, floating point programs has less computation involving linked data structures.

6. Conclusions and Future Work

A few pertinent conclusions can be drawn from the experiments conducted, the statistics collected and the observations made. The results of the experimental work can be summarized as follows:

- Data accesses from different cache blocks are often accessed together in a cache block. There are often a large number of such related addresses being accessed in a computation window.
- Preliminary observations seem to suggest that along with these data addresses, a large number of loads of non-related addresses also seem to occur.
- These computation windows often many times during program execution, especially for integer benchmarks.

Based on these, we may conclude that there is value in packing data from contiguous dependent loads together in a data trace cache line. This would definitely reduce cache misses if the data in the trace lines could also be accessed along with the data in the data cache. Whether the data trace cache can be used in the billion-transistor chips to connect computation and data is still an unanswered question. More research needs to be done into this before a definitive conclusion can be made.

The work to be done in future is pretty clear, at least the short-term needs based on what has been uncovered by this paper. More research into why we have such a large number of loads that are not included in any cache lines is required. Also, a dynamic identification of useful cache' lines is needed, rather than the current manual methodology adopted by us. Also, the behavior when the number of elements in the trace cache line is restricted to 8 or 16 must be studied as well as the maximum computation window cycle length.

References

[1] Y.N.Patt, S.J.Patel, M. Evers,, D.H.Friendly and J.Stark. One Billion Transistors, One Uniprocessor, One Chip. In *companion CDROM to IEEE CS Press Book, "The Anatomy of a Microprocessor: A Systems Perspective."*, pages 51-57, Sep 1997.

[2] M.H.Lipasti and J.P.Shen. Superspeculative Microarchitecture for Beyond AD 2000. In *companion CDROM to IEEE CS Press Book, "The Anatomy of a Microprocessor: A Systems Perspective."*, pages 59-66, Sep 1997.

[3] A.Moshovos, S.E.Breach, T.N.Vijayakumar, G.S.Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181-193, Jun 1997.

[4] A.Moshovos and G.S.Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture"*, pages 235-245, Dec 1997.

[5] A.Roth, A.Moshovos and G.S.Sohi. Dependence-based prefetching for linked data structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115-126, Oct 1998.

[6] C-K.Luk and T.C.Mowry. Compiler based prefetching for recursive data structures. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222-233, Oct 1996.

[7] A.Roth and G.S.Sohi. Effective jump-pointer prefetching for linked data structures. Submitted to *ISCA 99*.

[8] D.C.Burger and T.M.Austin. The simplescalar toolset, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun 1997.