Simulation of False Sharing Events using Dynamic Binary Instrumentation

November 30, 2009

Author:	Stephan M. Günther, B.Sc.
Supervisor:	Prof. Dr. Arndt Bode
Advisor:	Dr. Josef Weidendorfer

Faculty of Informatics
Technische Universität München

Abstract

Caches are a substantial building block for modern computers. In general data stored in memory can not be accessed individually but only by means of blocks, also called lines. This poses a potential problem for parallel computers: If multiple processor cores access data elements close to each other, they may both be located within the same cache line. As a result the copy of one core is invalidated when the other core updates an element within this block. Since the whole shared memory can be accessed by all processor nodes of a system it might happen, that one node modifies a value in memory which invalidates the copy of the respective memory block in the cache of another processor. Such a situation does not serve the purpose of data exchange between processor nodes and is therefore called *false sharing*. Depending on the hardware involved, such events may pose a serious bottleneck for performance which is very hard to detect. This work describes an approach to estimate the performance impact caused by false sharing and helps to locate critical code regions using the DBI framework Valgrind [5]. By recording memory references of different threads and their characteristics an estimate for an upper bound of false sharing events is calculated. Based on this it is possible to estimate the performance impact and to point to the code locations which caused the memory references in question.

CONTENTS

I Introduction 3

II Approximations 3
- II-A Model and notation . . . 3
- II-B Definitions . . . 3
  - II-B1 True sharing (TS) event . . . 4
  - II-B2 False sharing (FS) event . . . 4
- II-C False sharing and two threads . . . 5
- II-D Extension to multiple threads . . . 5
- II-E True sharing . . . 7
- II-F Combining the estimates for false and true sharing . . . 7
- II-G Examples . . . 8
  - II-G1 Matching accesses of same size at one offset, Figure 3(a) . . . 8
  - II-G2 Matching accesses of same size at multiple offsets, Figure 3(b) . . . 8
  - II-G3 Multiple accesses to disjoint areas, Figure 3(c) . . . 8
  - II-G4 Multiple accesses to overlapping areas, Figure 3(d) . . . 8
  - II-G5 Access pattern of a ring buffer, Figure 3(e) . . . 9
  - II-G6 Overlapping references of varying size, Figure 3(f) . . . 9
- II-H Problems in sharpening the approximation for false sharing . . . 9
- II-I Considering barriers . . . 10
- II-J Estimate of the performance impact . . . 10

III Current state of implementation 11

IV Results 11
- IV-A Synthetic false sharing benchmark . . . 11
  - IV-A1 Store/Store . . . 11
  - IV-A2 Modify/Modify . . . 12
- IV-B Prime Sieve . . . 14

V Conclusion and future work 15

VI Documentation 15
- VI-A Pluto . . . 15
  - VI-A1 Instrumentation routine . . . 15
  - VI-A2 Data structure for statistics . . . 15
  - VI-A3 Logging mechanism . . . 16
  - VI-A4 Considering barriers . . . 17
  - VI-A5 Flushing the log buffers . . . 17
  - VI-A6 Output file format . . . 17
- VI-B Output parser . . . 18
  - VI-B1 Command line reference . . . 18
  - VI-B2 Class hierarchy and data structure . . . 18

References 19

I. INTRODUCTION

False sharing can occur in any environment where multiple processing elements work on a shared memory. Examples are current shared memory architectures which also include desktops, workstations and notebooks with multicore processors. Within this work we focus on this class of systems. However, false sharing is also a problem for distributed shared memory systems (DSM). In fact, DSM systems may be even more severely affected since the block sizes are not cache lines but pages which increase the chance of false sharing.
It is very hard to investigate how many false sharing events occur at runtime and to quantify the temporal overhead because there is currently no way to reliably track such events. Intel proposes to sample specific cache related performance counters at runtime and to create statistics how often multiple threads access the same cache line at different offsets [4]. This proposal is based on the assumption that threads accessing the same cache line at the same offset exchange data (true sharing) while different offsets may indicate false sharing. However, such sampling based results heavily rely on the hardware being used and on a given temporal ordering of the memory accesses. This may vary between different runs which makes the result difficult to reproduce. Furthermore not all processors support the required performance counters.
Due to the problems in regard to time we developed a model which does not consider time at all. The underlying idea is to group memory references of multiple threads into parallel sections which are encompassed by barriers. Afterwards we create a worst-case sequence of memory reference within such a section as it might happen in reality. This enables us to employ dynamic binary instrumentation (DBI). This report presents an approach to estimate the number of false sharing events based on the well-known DBI framework Valgrind [5]. A given parallel application is instrumented at runtime by a new Valgrind tool which we call *Pluto*. It records all memory references and specific characteristics like thread ID, offset, size and type (load/store). The memory references are grouped into coherency blocks. Pluto writes the collected data for all coherency blocks which have been accessed by at least to different threads to a text file. This file can afterwards be analyzed to estimate the number of false sharing events that might occur when running the application on a parallel system. Hotspots of false sharing can be located by ordering the coherency blocks according to the estimated false sharing numbers. By using additional information collected by Pluto the code positions which caused the memory references can be located. Alternatively the data structures which a given coherency block belonged to could be identified. By providing a list of potential code positions or data structures which might cause a severe number of false sharing events the developer can decide if it is worth to review the code sections in question.

The remainder of this report is outlined as follows: Chapter II gives a comprehensive introduction into false sharing and the approximations for their occurrence. The current state of implementation is shortly outlined in Chapter III. In Chapter IV first results based on our approach are presented. Planned modifications and extensions are outlined in Chapter V. The prototypal tools implemented are documented in chapter VI.

II. APPROXIMATIONS

First this chapter gives a short overview of the notation used. Afterwards it is defined which situations are referred to as false sharing and true sharing respectively throughout this report. Using this definition the following sections will present different estimates for the occurence of false and true sharing. These make up the basis to estimate the performance impact as explained in the last section.

A. Model and notation

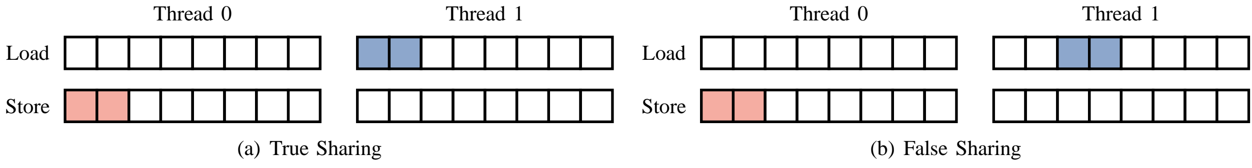
To be as independent as possible from any kind of machine architecture we consider only the size $|b|$ of a coherency block $b \in \mathcal{B}$ in our model. In case of shared memory systems b corresponds to a single cache line. Since we do not simulate caches in any way, we can divide the whole shared memory into a sequence of blocks b_i, $i \in \{0, 1, \ldots\}$. Let $\mathcal{T}$ be the set of thread IDs. Let further denote $\mathcal{X}$ the set of possible offsets and $\mathcal{Y}$ the set of allowed access sizes to memory. Then we can denote a single load or store operation by thread $t \in \mathcal{T}$ to coherency block $b \in \mathcal{B}$ with offset $x \in \mathcal{X}$ and size $y \in \mathcal{Y}$ by $l^{(t)}_{xy}[b]$ and $s^{(t)}_{xy}[b]$ respectively. The total number of load and store events with identical characteristics (coherency block, offset, size and thread) is denoted by $L^{(t)}_{xy}[b]$ and $S^{(t)}_{xy}[b]$ respectively. For example the total number of store operations $S^{(t)}[b]$ to block $b \in \mathcal{B}$ by thread $t \in \mathcal{T}$ is given by

$$S^{(t)}[b] = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} S^{(t)}_{xy}[b].$$

Sometimes the offset x and access size y are not considered. Then we rely only on the summed values $L^{(t)}[b]$ and $S^{(t)}[b]$.

B. Definitions

We turn now to definitions of false sharing and true sharing. In [2] different approaches to precisely define false sharing are discussed. However, a precise and practically applicable definition is difficult to find. For the scope of this report we give a



(a) True Sharing (b) False Sharing

definition of false sharing which is both reasonable and applicable but can not be mathematically expressed. However, this definition can be used in conjunction with our model to estimate what we define as false sharing which correlates with reality. For the following definitions we assume a concurrent section which is executed in parallel by multiple threads.

1) True sharing (TS) event:

We refer to a TS event if the exchange or invalidation of a coherency block serves the purpose of data exchange. Consequently one thread $t \in \mathcal{T}$ has to modify a coherency block $b \in \mathcal{B}$ at a given offset $x \in \mathcal{X}$ and of a given size $y \in \mathcal{Y}$ and at some point in the future a thread $t' \in \mathcal{T} \backslash \{t\}$ loads a value of size y in block b located at offset x.

Such an ordering of a store and a subsequent load operation is referred to as true haring. If another thread $t'' \in \mathcal{T} \backslash \{t, t'\}$ also loads a value in b located at x, y after t' did this is not counted as a second TS event. In reality a TS event might not be visible if the modified coherency block has been evicted due to a capacity miss before it is loaded by another thread. Since we do not simulate the cache we are not aware of capacity misses and will therefore always count the TS event, regardless how far away the subsequent load operation is. We do not consider the case where one thread stores data in blocks of size y while another thread loads data at size $y' \neq y$ as TS. Sequences of only load or store operations to b at x, y by different threads are neither considered true nor false sharing. The total number of TS events is denoted by θ. An ambiguous situation is shown in figure 1(a). Thread 0 writes to the first two elements within a coherency block while thread 1 reads the same elements. Note, that it still depends on the order of accesses if this is really true sharing. Assuming that each location marked in Figure 1(a) has been accessed exactly once by the corresponding thread we can differentiate between a number sequences. An incomplete list of such sequences is given as follows:

1) $s_{0,1}^{(0)} \rightarrow s_{1,1}^{(0)} \rightarrow l_{0,1}^{(1)} \rightarrow l_{1,1}^{(1)}$
   This sequence describes one TS event since thread 0 first writes data to the coherency block and afterwards the same data is read by thread 1.
2) $s_{0,1}^{(0)} \rightarrow l_{0,1}^{(1)} \rightarrow s_{1,1}^{(0)} \rightarrow l_{1,1}^{(1)}$
   The semantic outcome of this sequence is the same as for sequence 1). However, this leads to two exchanges of the cache line. The performace impact is the same as for false sharing. Nevertheless data is exchanged which gives two TS events according to the definition above.
3) $l_{0,1}^{(1)} \rightarrow l_{0,1}^{(0)} \rightarrow s_{1,1}^{(1)} \rightarrow s_{1,1}^{(0)}$
   In this case no data is exchanged since thread 1 reads the locations before thread 0 has written to them. If this sequence occurs at runtime it results in false sharing which is defined below.

2) False sharing (FS) event:

We refer to a FS event if exchange or invalidation of a coherency block does not serve the purpose data exchange. Consequently, one thread $t \in \mathcal{T}$ has to modify a coherency block $b \in \mathcal{B}$ at a given offset $x \in \mathcal{X}$ and of a given size $y \in \mathcal{Y}$ and at some point in the future a thread $t' \in \mathcal{T} \backslash \{t\}$ loads or stores a value of b located at offset $x' \in \mathcal{X} \backslash \{x\}$ or with size $y' \in \mathcal{Y} \backslash \{y\}$.

To fulfill the definition of an FS event it is therefore sufficient, that the subsequent access differs either in its offset or access size. One can argue here if accessing data at the same offset but of different size is FS or TS, but we currently define it as FS[1]. Sequences of only store operations to a block at the same offset and of the same size by different threads are treated as FS since overwriting results of another thread is no exchange of data. The total number of FS events is denoted by φ. An example is shown in Figure 1(b). In this case it is clear that no data have been exchanged since both threads accessed disjoint subsets of a coherency block. Nevertheless, the absolute number of false sharing events still depends on the sequence:

1) $s_{0,1}^{(0)} \rightarrow s_{1,1}^{(0)} \rightarrow l_{2,1}^{(1)} \rightarrow l_{3,1}^{(1)}$
   In this case thread 0 writes to both locations before thread 1 loads a value. This results in one FS event since the coherency block is exchanged only once.
2) $s_{0,1}^{(0)} \rightarrow l_{2,1}^{(1)} \rightarrow s_{1,1}^{(0)} \rightarrow l_{3,1}^{(1)}$
   Here the accesses of both threads are interleaved which results in three exchanges of the cache line. This gives therefore

[1]For example copying an array of non-character elements might be done byte-wise which would result in false sharing if another thread has written the elements before.

a total of three FS events.

Given these examples it is clear that the global temporal order of memory references plays an important role for the number of FS and TS events in a multi-processor system. Unfortunately this order is not unique in reality and can vary in different runs of the same threaded code. For example, it is not guaranteed that both threads start working at the same time. Therefore one thread may hurry ahead some cache lines within a data structure which may lead to a lower number of FS events compared to the situation where both threads had started at exactly the same time. In general the order of memory references between different threads within a parallel section is not predictable. Instrumenting an application using dynamic binary instrumentation completely destroys any temporal relations between threads for two reasons. First, injected code interferes at runtime and changes when memory references are made. Second, Valgrind serializes parallel applications which of course eliminates any concurrency. Only the sequence of references per thread can be obtained. Although this might be used to refine estimates in theory it is not feasible in practice for reasons discussed in Section II-H. For this reason we try to construct a sequence of memory references given their absolute numbers which maximizes the respective type of events. The estimated absolute values for the number of FS and TS events are denoted by $\tilde{\varphi}$ and $\tilde{\theta}$ respectively.

C. False sharing and two threads

Due to the lack of reliable information regarding time we try to construct a worst case sequence of memory references which maximizes the number of FS events within a parallel section. Assuming only two threads $\mathcal{T} := \{0, 1\}$ a first estimate can be made by only considering the absolute number of accesses per thread on each coherency block $L^{(t)}[b]$ and $S^{(t)}[b]$ with $t \in \{0, 1\}$ and $b \in \mathcal{B}$. For now we do not consider offset and size of references. This makes it impossible to differentiate between TS and FS at all. However, if TS is counted as FS we increase the estimate for a bound which might result in a big error but in general does not underestimate the number of FS events. Given a number of memory references the worst case is an alternating sequence of store and load operations by thread 0 and 1 respectively. After there are no suitable operations left we continue with stores of thread 1 and loads of thread 0. Finally there might be store operations of both threads left which can also be arranged in a sequence to increase the number of FS events. This can be written as sequence of non-distinguishable operations:

$$s^{(0)} \to l^{(1)} \to s^{(0)} \to l^{(1)} \to \ldots \to s^{(1)} \to l^{(0)} \to s^{(1)} \to l^{(0)} \to \ldots \to s^{(0)} \to s^{(1)} \to s^{(0)} \to s^{(1)} \to \ldots$$

The first reference causes no FS since it is a compulsory miss and any subsequent access causes one FS event. We neglect compulsory misses since there is at most one per coherency block accessed. It is important to consider the combinations of loads and stores first because solely load operations can not cause false sharing. Only if there are store operations of more than one thread left and there are no more store operations available we construct a sequence of solely store operations. The problem of constructing such a worst case sequence can also be seen from the point of set-theory. Since we do not consider time we can also abandon the view of sequences. Instead we can assume sets of load and store operations for both threads. Now we combine each load of thread 0 with a store of thread 1 and vice versa. Afterwards there might by store operations left for both threads. These can also be combined. By counting the number of combined operations we end up with a value which equals in length to the sequence above. From both views we can derive the following formula which gives an estimate of the FS events $\tilde{\varphi}$ in the worst case:

$$\tilde{\varphi}[b] = 2 \cdot \left[\underbrace{\min\left(S^{(0)}[b], L^{(1)}[b]\right)}_{=:\alpha} + \underbrace{\min\left(S^{(1)}[b], L^{(0)}[b]\right)}_{=:\beta} \right] + 2 \cdot \underbrace{\max\left(\min\left(S^{(0)}[b] - L^{(1)}[b], S^{(1)}[b] - L^{(0)}[b]\right), 0 \right)}_{=:\gamma} \quad (1)$$

The part abbreviated by α considers all possible combinations of store operations by thread 0 with load operations by thread one. Accordingly β considers the inverse case. Remaining store operations (if any) of both threads are combined by γ. If $L^{(1)}[b] < S^{(0)}[b]$ and $L^{(0)}[b] < S^{(1)}[b]$ hold true, there are remaining store operations to construct a sequence where thread 0 and thread 1 write to b in an alternating fashion. If either $L^{(1)}[b] \geq S^{(0)}[b]$ or $L^{(0)}[b] \geq S^{(1)}[b]$ holds true, γ is forced to 0. Equation (1) does a perfect job if there are no TS events in reality. Unfortunately it can not differentiate between FS and TS at all which is why the result may turn out to be a massive overestimation. Nevertheless it may be useful if the programmer knows where data exchange between threads is performed. If Pluto using (1) comes up with code positions where no data is exchanged the programmer also knows that there is a potential source of false sharing.

D. Extension to multiple threads

It is not straight forward to find a closed form like (1) for more than two threads. The problem consists in

- updating the numbers of remaining loads and stores after considering a tuple $(L^{(u)}, S^{(v)})$ for $u, v \in \mathcal{T}$, $u \neq v$ and
- the order in in which the tuples are processed.

```
phi = 0;
sort( l );
sort( s );

while( exists (l[u] > 0) and (s[v] > 0) for (u,v) in {TxT} and u != v ) do
   for all i in 1..|T| do
      for all j in 1..|T| do
         if thread( l[j] ) != thread( s[i] )
            tmp = min( l[j], s[i] );
            l[j] -= tmp;
            s[i] -= tmp;
            phi += tmp;
            goto _next;
         fi
      done
   done

   _next:
   sort( l );
   sort( s );
done

for all i in 1..|T| do
   for all k in i..|T| do
      if thread( s[j] ) != thread( s[i] )
         tmp = min( s[j], s[i] );
         s[j] -= tmp;
         s[i] -= tmp;
         phi += tmp;
      fi
   done
done

return phi;
```

Listing 1. Calculating $\tilde{\varphi}[b]$ for $|\mathcal{T}|$ threads

The first issue only prevents us from giving a closed form like (1) since notation becomes very tedious and confusing. The second problem is more severe because the order in which loads and stores of different threads are combined can lead to different results. Consider the following example:

- $L^{(0)} = 50$, $L^{(1)} = 5$, $L^{(2)} = 100$
- $S^{(0)} = 50$, $S^{(1)} = 0$, $S^{(2)} = 100$

We can now calculate $\tilde{\varphi}$ in two different ways:

$$\tilde{\varphi}_1 = 2 \cdot 50 + 2 \cdot 5 + 2 \cdot 45 = 200$$
$$\tilde{\varphi}_2 = 2 \cdot 50 + 2 \cdot 50 + 2 \cdot 5 = 210$$

We might first combine 50 loads of thread 0 with 50 stores of thread 2 leaving another 50 stores of thread 2. Afterwards we take the 5 loads of thread 1 and combine them with 5 stores of thread 0. Finally we take the remaining 45 stores of thread 0 and combine them with loads of thread 2. This makes a total of 200 possible FS events. However, we could also first combine loads of thread 2 with the 50 stores of thread 0. Afterwards there are still 5 loads of thread 1 left which can be combined with stores of thread 2. This would results in a total of 210 FS events. A priori it is unclear in which order we have to combine events to maximize $\tilde{\varphi}$. The idea is now to combine large values first and to keep small values as long as possible. Therefore, we keep all possible combinations between different threads open as long as possible. To accomplish this we sort the loads and stores by their value in descending order. Afterwards we start to combine the largest values of different threads and sort the vectors again if necessary. Note, that at most one element has to be sorted after each step. Finally we might end up with a situation where stores of more than one thread but no more loads are left. In this case we combine stores of different threads until only stores of a single thread are left. A formal proof that this algorithm really gives the maximum value for $\tilde{\varphi}$ can not be given at the moment. Listing 1 outlines an algorithm in pseudo code which calculates the estimate for a single coherency block. At first $\tilde{\varphi}$ is initialized with zero and the vectors $\boldsymbol{l}$ and $\boldsymbol{s}$ are sorted by size. Afterwards we iterate over these vectors as long as there exist components in both vectors which are non zero and represent memory references of different threads. The inner loops make sure that we test all possible combinations and check if the components denote references of different threads. If this is the case we combine these values and update the vectors and the estimate. Afterwards we leave the inner loops, sort the vectors and start probing at the beginning of both vectors again. If no possible combinations are left we start combining left store operations by different threads. In this case the order is no longer relevant.

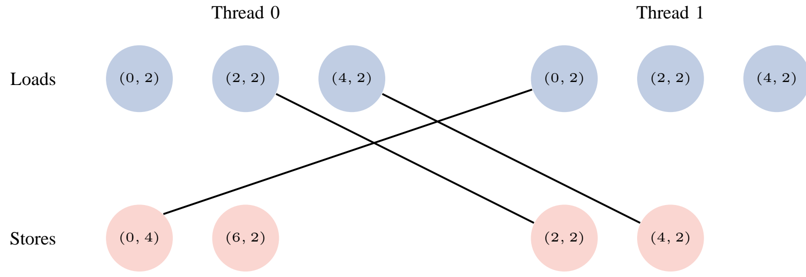


Fig. 1. Finding pairs of matching load and store operations. The tuple (x, y) denotes the offset $x \in \mathcal{X}$ and size $y \in \mathcal{Y}$ of each memory reference.

E. True sharing

Now we derive an estimate for the upper bound of TS events that might occur given a best case sequence of operations. This estimate $\tilde{\theta}$ may be useful to make a statement regarding the reliability of $\tilde{\varphi}$ calculated according to (1). Again we assume only two threads and no temporal order of the memory references. To determine if two memory references might result in a TS event we have to consider their size and offset. This means, if we find a pair $(s_{xy}^{(u)}[b], l_{xy}^{(v)}[b])$ of references where $u \neq v$ and $u, v \in \mathcal{T}$ this might have been a TS event in reality. This process is illustrated by Figure 1 for a single coherency block. Two loads of thread 0 match with two stores of thread 1 in size and offset which we count as a total of four TS events. In a similar way one load of thread 1 matches a store of thread 0 which is counted as two TS events. The justification for doubling the number of events is that for a given sequence of matching store and load operations an eviction of the modified line is triggered when a value is loaded and the copy at the reader is invalidated when a value is modified again. If $\tilde{\theta}$ turns out to be more than an order of magnitude smaller than $\tilde{\varphi}$ it is very likely that FS occurs, unless there are some temporal constraints in reality which prevent FS at all. If on the other hand $\tilde{\theta} \approx \tilde{\varphi}$ holds then we can not make any statement regarding the reliability. The estimate $\tilde{\theta}[b]$ for a given coherency block $b \in \mathcal{B}$ can be calculated as follows:

$$\tilde{\theta}[b] = 2 \cdot \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \left[\min\left(S_{xy}^{(0)}[b], L_{xy}^{(1)}[b]\right) + \min\left(S_{xy}^{(1)}[b], L_{xy}^{(0)}[b]\right) \right] \tag{2}$$

The inner term of the sums is very similar to $\alpha + \beta$ in (1), except for the added offsets and sizes. Since alternating store operations of both threads can not serve the purpose of data exchange, these are not considered as true sharing and are therefore omitted in (2). It is possible to extend this estimate to more than two threads in a very similar way as described in Section II-D.

F. Combining the estimates for false and true sharing

The first estimate $\tilde{\varphi}$ for FS does not consider offset and size of memory references. Therefore, it is not able to differentiate between TS and FS at all. Now that we have an estimate for the number of TS events, it appears tempting to obtain a better estimate

$$\tilde{\varphi}' := \tilde{\varphi} - \tilde{\theta}. \tag{3}$$

Although this approach delivers astonishing exact results in first tests as we will see in Section IV-B, these might be more the result of coincidence. The general problem here is, that the validity of the result completely depends on the unknown sequence of memory references. Consider the example in Figure 2. In this case $\tilde{\varphi} = \tilde{\theta}$ holds wherefore $\tilde{\varphi}'$ vanishes. However,

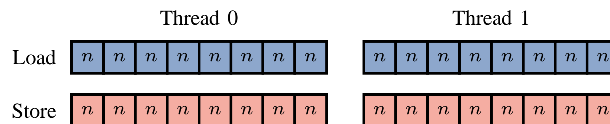


Fig. 2. Access pattern with unknown sequence might misslead $\tilde{\varphi}'$

given this access patterns and the same number n of references to each cell we can also construct the worst case sequence of references. Therefore $0 \leq \varphi \leq \tilde{\varphi}$ probably holds. Unfortunately we can not make any further statement about the true value for φ without knowledge of the actual sequence of references per thread. However, it is arguable if a meaningful real-world application would produce an access pattern as given with exactly the same number of references and nevertheless produces a significant number of FS. For real-world scenarios such an access pattern is a very strong indication that both threads work together on a common data set. Furthermore there are probably some kind of synchronization mechanisms implemented to avoid race conditions. Eventually detecting and handling these synchronization points might be the key to finally decide if $\tilde{\varphi}'$ gives a meaningful result. This is an open question at the moment.

G. Examples

We will now investigate some different access patterns of two threads and compare the values of $\tilde{\varphi}$, $\tilde{\theta}$ and $\tilde{\varphi}'$. Furthermore we will discuss for each example wether the estimates reflect what we would expect from an application producing a similar access pattern. For ease of notation we assume only a single memory block $b \in \mathcal{B}$ and therefore omit the index. Furthermore we consider only references within a single parallel section.

1) Matching accesses of same size at one offset, Figure 3(a):
Thread 0 writes to the first position n times while thread 1 loads this position just as often. Given such an access pattern we would expect TS, since size, offset and also the number of accesses matches between threads. For example thread 0 might store some kind of status value at this position which is periodically read by thread 1 and triggers some kind of action. A synchronization between both threads is not obligatory. The estimates evaluate to

$$\tilde{\varphi} = 2n,\ \tilde{\theta} = 2n,\ \tilde{\varphi}' = 0.$$

Given only the total numbers we cannot make any meaningful statement regarding the actual number of FS events. However, by looking at the access pattern it is at once clear that there con not be any FS. It is unsure if $\tilde{\theta} = \theta$ is really fulfilled, but since there is only a single class of loads and a corresponding class of stores FS can never happen. For this reason $\tilde{\varphi}'$ is probably correct.

2) Matching accesses of same size at multiple offsets, Figure 3(b):
This case is very similar to the first one but this time there are multiple offsets. From a real world application we would again expect TS. Synchronization is still not obligatory. The estimates give:

$$\tilde{\varphi} = 4n,\ \tilde{\theta} = 4n,\ \tilde{\varphi}' = 0$$

The conservative result is probable $\tilde{\varphi} = 4n$ since this time we can in fact construct a worst case sequence of this length. Calculating the overhead with this estimate can easily lead to the claim, the application would suffer from an overhead which is an order of magnitude higher than the total runtime of the application. On the other hand $\tilde{\varphi}'$ corresponds to what we expect from a real world application producing this access pattern. However, we can not know it for sure without knowing both the sequence of accesses of each thread and their temporal interleaving. Note that such an access pattern is far from optimal even in case of TS. Assume that thread 0 writes the first position, then thread 1 reads it, afterwards thread 0 writes the second position which is also read from thread 1. Now thread 0 starts to write to the first position again. This corresponds to a ring buffer of size 2. It would be more efficient to place both items in different cache lines to avoid frequent exchanges of the cache line. However, detecting such a situation definitely requires knowledge about the sequence of references and is out of the scope of this report.

3) Multiple accesses to disjoint areas, Figure 3(c):
In this case two threads access disjoint areas of the same memory block. Both load and store operations may happen. The actual number of references per thread does not have any impact on the qualitative statement of this example. For simplicity all positions are assumed to be accessed n times (load and store). This gives the following values for the estimates:

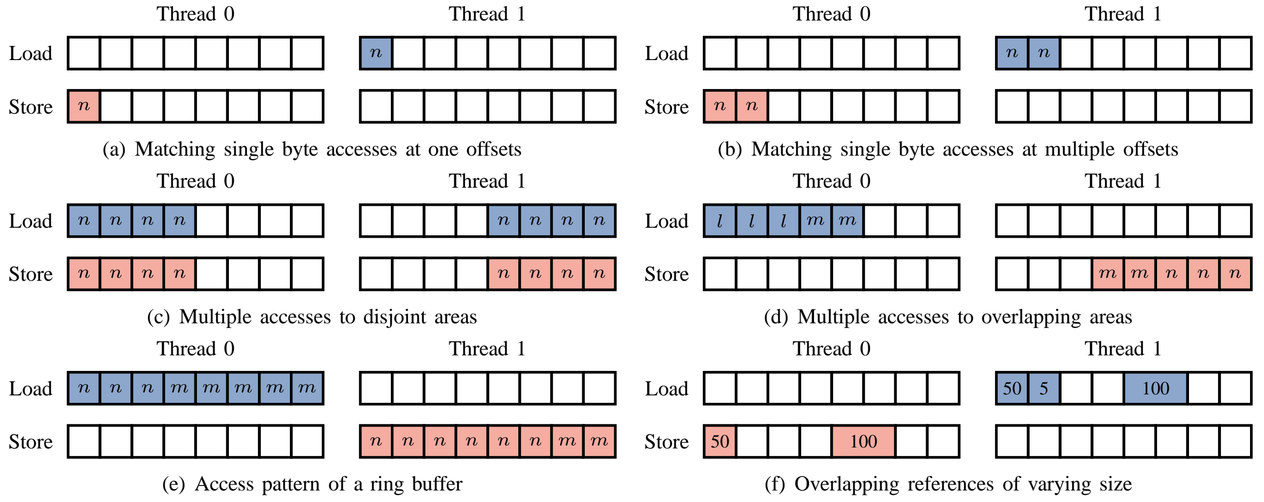
$$\tilde{\varphi} = 16n,\ \tilde{\theta} = 0,\ \tilde{\varphi}' = 16n$$

Without knowledge of time it is not sure, if the FS estimates are tight. However, since there cannot be any TS in this example we know that this access pattern potentially leads to a very high number of FS events. Assuming the worst case is the best we can do at this time. This access pattern is common for real world applications. Assume an array of size N which is executed in parallel by two threads. If the array element with index $N/2$ is not located at the beginning of a cache line a similar situation would arise. However, there is only a single cache line affected and only a small portion of the operations would be performed on the corresponding memory block. Therefore the overhead incurred by FS will go to zero for sufficiently large arrays. For small arrays it would be severe. Both cases are perfectly modeled by the estimates.

4) Multiple accesses to overlapping areas, Figure 3(d):
Assume that for some reason one thread only loads from a sequence of memory locations while another thread only writes to another sequence of locations and that there exists an overlapping section. Within this overlapping TS seems likely. However, FS can occur due to the disjoint parts. Given varying number of references for all three areas the estamites evaluate to:

$$\tilde{\varphi} = 4m + 6 \cdot \min(l, n),\ \tilde{\theta} = 4m,\ \tilde{\varphi}' = 6 \cdot \min(l, n)$$

This time there is a difference between $\tilde{\varphi}$ and $\tilde{\varphi}'$ depending on the number of references. Without further information we cannot be sure which one is more applicable. However, matching offsets and sizes of memory references by different threads to the same code location are an indicator for TS. Therefore, $\tilde{\varphi}$ may be more adequate for real world scenarios although this is based only an assumption and easy to disprove by giving a suitable sequence of memory references leading to this access



(a) Matching single byte accesses at one offsets
(b) Matching single byte accesses at multiple offsets
(c) Multiple accesses to disjoint areas
(d) Multiple accesses to overlapping areas
(e) Access pattern of a ring buffer
(f) Overlapping references of varying size

Fig. 3. Different examples for access patterns.

pattern.

5) Access pattern of a ring buffer, Figure 3(e):
For $m := n-1$ this examples models the access pattern of a ring buffer. Thread 1 inserts elements into the buffer while Thread 0 extracts them. Given this example the buffer would currently contain three elements (offsets 3,4,5). To indicate if the ring buffer currently contains any elements a synchronized variable is needed which might be located anywhere else in memory. Calculating the bounds gives:

$$\begin{aligned}
\tilde{\varphi} &= 2 \cdot \min(3n + 5m, 6n + 2m) \\
&= 6 \cdot \min(m, n) + 4m + 6n \\
&= 16n - 10, \quad \text{for } n \geq 1 \\
\tilde{\theta} &= 6n + 4m + 6 \cdot \min(m, n) \\
&= 16n - 10, \quad \text{for } n \geq 1 \\
\tilde{\varphi}' &= 0
\end{aligned}$$

Which bound is tight depends now on the definition of FS. Since the ring buffer definitely serves the purpose of data exchange, this example does not give any FS events according to our definition. Therefore the outcome of $\tilde{\varphi}'$ is tight. However, in terms of avoiding frequent cache line exchanges it would be much more efficient to either write elements in neighboring memory blocks or to force the producer to complete a full cache line before the consumer is allowed to read the first element. Given such an implementation $\tilde{\varphi}'$ remains tight while $\tilde{\varphi}$ would be the wrong choice.

6) Overlapping references of varying size, Figure 3(f):
Finally we consider an example with different access sizes. Although there are matching operations which might result in TS, there are tuples of load and store operations of different sizes. For examples two different variables might be frequently exchanged between two threads. The first one resides at offset 0 while the second one is located at offset 4. It depends on the temporal order if this results in FS or not. The bounds evaluate to:

$$\tilde{\varphi} = 300, \ \tilde{\Theta} = 300, \ \tilde{\varphi}' = 0$$

Assuming the worst case $\tilde{\varphi}$ is most likely an overestimation of FS events. However, taking $\tilde{\varphi}'$ is now likely an underestimation. It depends on time again which estimate is better suited.

H. Problems in sharpening the approximation for false sharing

Assume that the first five positions of a coherency block are just as often read by thread 0 as written by thread 1. This corresponds to Figure 3(d) where $l = m = n$ holds. The maximum number of TS events that can occur is $\tilde{\Theta} = 4n$. However, it is not sure that the overlap really indicates TS events. For example thread 0 might read n times from offset 0 while thread 1 writes n times to offset 3 which would result in $2n$ FS events. Depending on the remaining sequence of accesses the number of FS events might further increase. Therefore, the estimate $\tilde{\varphi}$ is the best possible estimate for this case which is definitely correct but also very likely an overestimation. Of course it is also not sure that FS occurred at all. For example all load operations

by thread 0 might happen before the first store of thread 0 or vice versa. However, we can not know this given only absolute numbers of references without temporal context.
A possible approach to sharpen $\tilde{\varphi}$ is to take the size and offset of references somehow into account. Consider again the situation in Figure 3(f). We could construct a worst case sequence to maximize FS with considering size and offset of access in two ways:

1) At first thread 0 writes to offset 0 while thread 1 reads from offset 1. This gives 10 FS events and 45 left store operations at offset 0. Thread 0 continues to write to offset 0 while thread 1 now loads from offset 4. This gives another 90 FS events and 55 left loads at offset 4. Now thread 1 turns to load from offset 0 and thread 0 write to offset 4 which gives another 100 FS events and therefore a total of 200 FS events.
2) Now we construct a slightly different sequence. Thread 1 starts loading from offset 4 and thread 0 writing to offset 0. This gives 100 FS events. Afterwards thread 0 writes to offset 4 while thread 1 loads from offset 0 and 1. This gives 110 FS events and therefore a total of 210 FS events.

The example above shows that the sequence in which operations are combined does matter. Given more access size and offsets, possibly even more than two threads, this problem becomes even more difficult. It might be some kind of combinatorial optimization problem. So far we found no way to formalize the process of finding the worst case sequence with considering access sizes and offsets even for only two threads.

I. Considering barriers

Barriers can be quite easily considered as long as they affect *all* threads. Synchronization between a subset of threads is much more difficult. For now we assume a number of *parallel sections* defined as a code segment which can be executed in parallel and independently by an arbitrary number of threads and which is encompassed by barriers. Anything discussed so far holds within a single parallel section. Subsequent parallel sections can be taken into account by introducing an index s to the set of coherency blocks $\mathcal{B}$ where s denotes the parallel section. Let $\mathcal{S} := \{0, 1, \ldots, |\mathcal{S}| - 1\}$ be the set of parallel sections and $\mathcal{B}_s$ the set of coherency blocks which have been accessed within section $s \in \mathcal{S}$. We note that at most one FS (or TS) event can occur between subsequent parallel sections. This happens when a core accesses a memory block the first time within this section which has been accessed by another core in the previous section provided that no compulsary miss occurred in the meantime. The number of FS (TS) events between any section s and its subsequent section $s + 1$ is bounded above by $|\mathcal{B}_s \cap \mathcal{B}_{s+1}|$ if $s + 1 \in \mathcal{S}$ and 0 otherwise. Estimates $\tilde{\varphi}$ and $\tilde{\theta}$ considering multiple parallel sections can be expressed by $\tilde{\varphi}_s$ and $\tilde{\theta}_s$ per parallel section $s \in \mathcal{S}$:

$$\tilde{\varphi} = \sum_{s \in \mathcal{S}} \tilde{\varphi}_s + \sum_{s=0}^{|\mathcal{S}|-2} |\mathcal{B}_s \cap \mathcal{B}_{s+1}| \tag{4}$$

$$\tilde{\theta} = \sum_{s \in \mathcal{S}} \tilde{\theta}_s + \sum_{s=0}^{|\mathcal{S}|-2} |\mathcal{B}_s \cap \mathcal{B}_{s+1}| \tag{5}$$

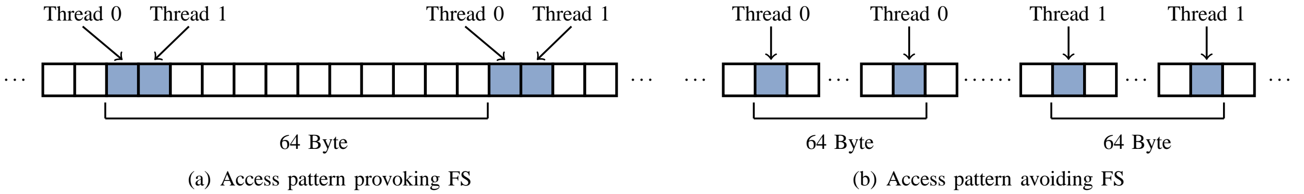
Note that considering parallel sections is not necessary if we are only interested in $\tilde{\varphi}'$ because the influence of barriers vanishes for this case.

J. Estimate of the performance impact

So far we were only concerned in giving an estimate for the number of FS events. However, the absolute number is not meaningful for a quick decision if the incurred overhead by FS is severe. For this reason we convert the absolute numbers into a temporal overhead Δt by multiplying the corresponding estimate with the approximate penalty per FS event. Let τ_c be the overhead per FS event measured in clock cycles and f be the core frequency measured in Hz of the processors for which the overhead should be calculated. Given the most simple bound $\tilde{\varphi}$ the overhead is given by

$$\Delta t := \frac{\tilde{\varphi} \cdot \tau_c}{f}, \quad [\Delta t] = \mathrm{s} \tag{6}$$

This also holds true for $\tilde{\varphi}'$ but the calculated overhead might be an underestimation for this case. There are two problems regarding this calculation. First it might give a value which is larger than the total runtime of the application. This is basically the result of a possible overestimation of FS events. Second, this approach assumes a fixed penalty per FS event which is in general not true. The penalty depends on the processor cores between the FS event happens. Given a dual processor system equipped with dual core processors where both cores share a common L2 cache the penalty might be around 30 clock cycles if the FS event occurs between cores on the same socket but about 250 cycles if it occurs between cores on different sockets. This difference is almost an order of magnitude and makes the temporal estimate inaccurate. Since we cannot know on which core a thread would be executed in reality there is no solution to this problem. Therefore, the programmer is asked to supply the core frequency and a penalty of a system, for which the overhead should be calculated. If no penalty is given, a modest penalty of 50 cycles is currently assumed.



(a) Access pattern provoking FS

(b) Access pattern avoiding FS

Fig. 4. Access patterns to provoke or avoid FS. The type of access depends on the selected mode.

III. Current state of implementation

At this time the Valgrind tool Pluto is responsible only for collecting statistical information about memory references. This is done by logging the absolute number of references $S^{(t)}_{x,y}[b_s]$, $L^{(t)}_{x,y}[b_s]$ for each parallel section $s \in \mathcal{S}$. To reduce the output Pluto only logs coherency blocks which are accessed by at least two threads within a single parallel section. Note, that this makes it impossible to consider the boundary between parallel sections as described in II-I since sharing between parallel sections can also occur if a block has been accessed only by a single thread within a section. For this reason we currently take the boundary between parallel sections not into account which might lower $\tilde{\varphi}$ and $\tilde{\theta}$ at most by the number of blocks accessed within a parallel section while $\tilde{\varphi}'$ remains exact. This has no influence at all on the examples discussed in Chapter IV since these examples rely on a single parallel section only. The logfile produced by Pluto is afterwards read by a parser which allows to interactively display summaries about coherency blocks and details of a single block including access masks, the number of references by different threads and code positions which caused the memory references. Furthermore it calculates the estimates $\tilde{\varphi}$, $\tilde{\theta}$ and $\tilde{\varphi}'$ and allows to order the coherency blocks according to the FS estimates. By supplying the core speed and penalty per FS event the total numbers can be converted in a temporal penalty Δt.

IV. Results

This chapter discusses first results of our appraoch. For this purpose we created two benchmarks. The first one is synthetic and specially designed to point out the overhead incurred by FS. The second benchmarks calculates prime numbers and suffers from FS after parallelization.

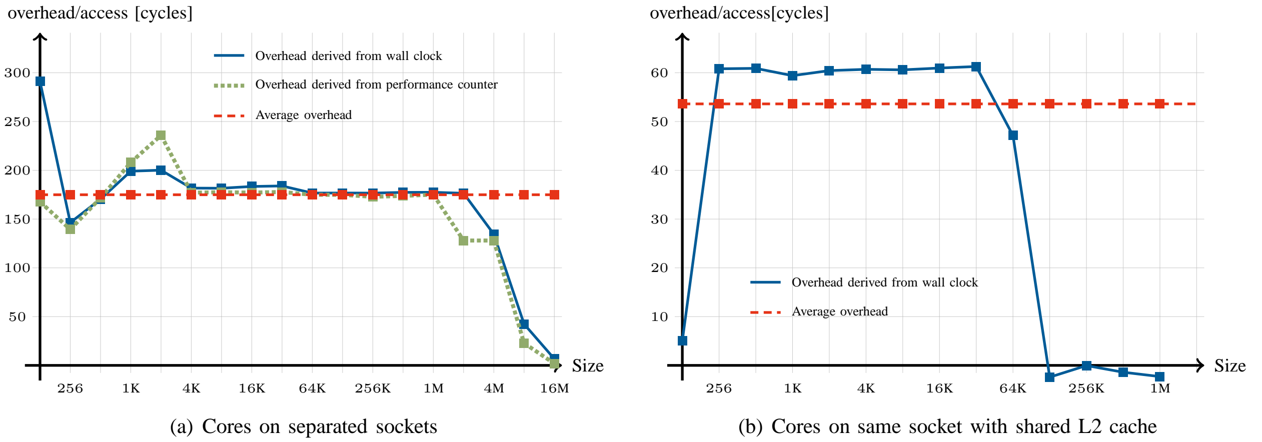
A. Synthetic false sharing benchmark

The synthetic FS benchmark is designed to trigger as much FS events as possible. Two threads traverse an array in 64 Byte strides with 4 byte offset to each other. This is illustrated by Figure 4(a). Thread 0 accesses an element of four byte at offset 0 while thread 1 accesses a value at offset 4 of the same size. Afterwards both threads jump 64 byte ahead which corresponds to the next cache line. The threads are pinned to processor cores. Therefore, each core accesses a cache line exactly once. The type of access depends on the selected mode. Available combinations for both threads are store/store, modify/modify and store/load. This is repeated for a given number of iterations. The benchmark is designed in a way such that the number of memory accesses per thread remains constant independently of the array size which makes the results easier to understand. To accomplish this the number of iterations is adapted dynamically. Since the number of memory references remains constant the estimates $\tilde{\varphi}$ and $\tilde{\theta}$ are also constant. This is the reason why the overhead as given by Pluto reduces to a trivial case. However, this benchmark can be used to determine the penalty per FS event for a given system which can be passed to Pluto to modify its preset default penalty. To obtain the overhead induced by FS through measuring there exists a second variant of this benchmarks which avoids FS at all by performing the same memory references on disjoint parts of the array as illustrated in Figure 4(b). Note, that this doubles the total working set. This does not matter as long as the processors executing the benchmark do not have a common cache level, e.g. processors located on different sockets. If they have a common cache level, the results are only reliable up to half of the L2 cache size since for larger arrays capacity misses are inevitable. Results for different access types are discussed below.

1) Store/Store:

We executed the benchmark with varying array sizes from 128 byte to 16MiB with FS enforced or avoided. The threads were pinned to processor cores located on different sockets which makes the penalty per FS event very high. We did a total of $6,25 \cdot 10^7$ memory references per run. For the FS estimate holds $\tilde{\varphi} = 1.25 \cdot 10^8$ events according to (1). The estimate given by Pluto is slightly higher (around 5000 additional FS events) since both threads do not only access the working set but are also synchronized at the end of the test loop by a barrier. This additional overhead is more than four orders of magnitude smaller than the analytical estimate and therefore insignificant. For this example we know that no TS can occur. In fact, $\tilde{\varphi}$ evaluates to zero for the given access pattern. Since the number of memory references is known exactly we can derive the number of clock cycles per memory reference by using the wall clock time:

$$c := \frac{\text{time needed for execution}}{\text{number of memory references}} \cdot \text{core frequency [Hz]}$$



(a) Cores on separated sockets

(b) Cores on same socket with shared L2 cache

Fig. 5. Overhead per memory access incurred by FS for store/store operations, approximately corresponds to the penalty per memory reference.

The overhead incurred by FS per memory reference can be calculated as delta between c for runs with and without FS since we know that every memory access should trigger an FS event. The results are illustrated in Figure 5(a). The continuos line denotes the overhead per memory reference in clock cycles. Besides some fluctuations at very low array sizes we get an almost constant overhead up two an array size of 4MiB. At this point the L2 cache begins to fill up. The overhead reduces significantly at a size of 8MiB and converges to zero for larger arrays. The reason for this consists in capacity misses of the last exclusive cache level per core. This means that a modified cache line of one core is likely to be evicted due to a capacity miss before it is requested by the other core. The average overhead between 128 byte and 2MiB lies around 150 clock cycles. For lack of exact knowledge we will accept this as penalty τ_c per FS event for now. This value is supplied to Pluto to translate the estimate $\tilde{\varphi}'$ to a temporal overhead. Since the FS estimate is tight in this example and the axis of ordinates is given as FS Overhead per memory reference in cycles Pluto's output denoted by the dashed line results in a constant and reflects the average penalty τ_c per clock cycle as long as not capacity misses occur. When the array size exceeds the L2 cache size Pluto's estimate still remains constant since we do not consider the finite cache size at the moment. To verify the results obtained by time measures we repeated the test and read the performance counters by using pfmon. Since the access pattern is identical for both cores we measured the events `L2_LINES_OUT:ANY` and `L1D_CACHE_ST:MESI` of a single core with and without FS. Afterwards we calculate ratio between L2 evictions and the total number of memory references to the L1 cache

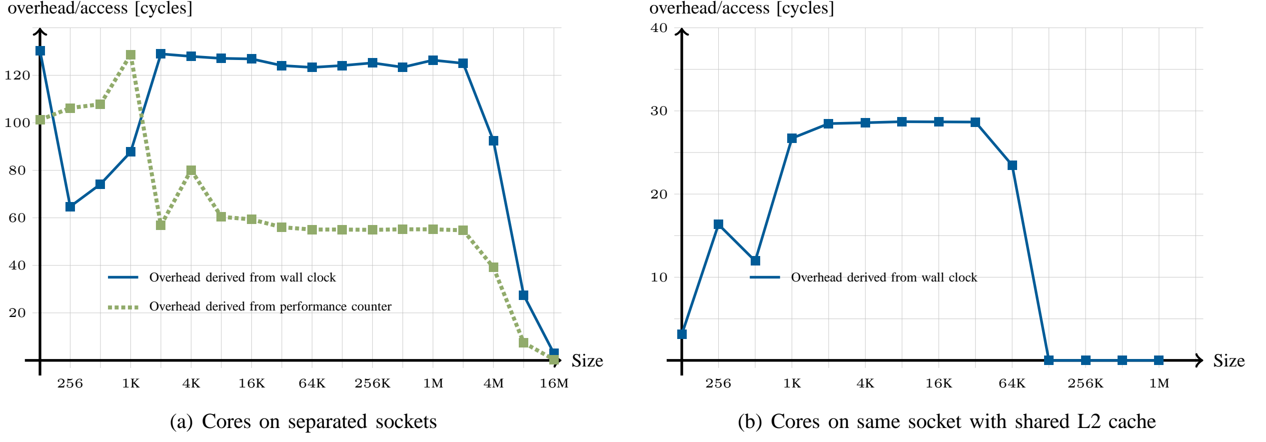
$$\rho := \frac{\texttt{L2_LINES_OUT:ANY}}{\texttt{L1D_CACHE_ST:MESI}}.$$

In this case FS $\rho \approx 1$ holds at array sizes smaller than the L2 cache. This proofs that each memory reference triggered an L2 eviction at the other core and vice versa. By calculating the difference between the ratios for a run with and without FS and multiplying the result with the estimated penalty we end up with the dashed line in Figure 5(a). This correlates very closely to the overhead measured using the wall clock method.

In a second test we pinned the threads to cores located on the same socket and with common L2 cache. The remaining setup is the same as for the previous test. Figure 5(b) shows the results. The continuos line denotes again the overhead as obtained by the wall clock approach. As the array size exceeds the L1 cache size (32KiB) the overhead incurred by FS drops to zero within error of measurement. This is clear since modified cache lines can be exchanged over the common L2 cache. The dashed line shows again the estimate by Pluto for $\tau_c = 47$ cycles which basically reflects the average overhead within the L1 cache. At this point the importance of the penalty factor τ_c for the given situation becomes clear. If Pluto uses the penalty as obtained by the test on different sockets in this example the overhead would be overestimated by approximately a factor of 3. Therefore, τ_c is a system dependent factor and must be supplied to Pluto. Otherwise we have to accept inaccuracies induced by a preset value for the penalty. This example also shows the dependence of FS on the cache size. These must not be to neglect at least for such regular access patterns. At the moment it is unknown if this is also a problem for real-world applications.

2) Modify/Modify:

The setup remains the same as for store/store but modifying implies a load and a subsequent store operation. The number of memory references per thread is therefore doubled. However, it is no longer possible that each memory reference causes an FS event. Therefore, the penalty per memory reference does not longer reflect the penalty per FS event like in the store/store example. The results for cores located on different sockets are given in Figure 6(a). Since the result of Pluto would show again a constant at approximately the supplied value τ_c it is omitted. It is more interesting here to investigate the penalty per FS event. The overhead per memory reference obtained by the wall clock method is shown by the continuous line in Figure 6(a). The drop at arrays between 256 and 1024 byte can not be explained. If we foresee this unclarity the average overhead lies



(a) Cores on separated sockets

(b) Cores on same socket with shared L2 cache

Fig. 6. Overhead per memory access due to FS for modify/modify operations.

Array size	L2_LINES_OUT	L1D_CACHE_LD:MESI	L1D_CACHE_ST:MESI	ρ
128 byte	61026929	3953976	62506207	0.92
256 byte	60724640	513642	62505377	0.96
512 byte	61162812	30499	62505876	0.98
1 KiB	74028725	902592	62506186	1.17
2 KiB	64464253	62520950	62506394	0.52
4 KiB	90907380	62533226	62553776	0.73
8 KiB	68582463	62622073	62503795	0.55
16 KiB	67296598	62519515	62503674	0.54
32 KiB	63709466	62756873	62548383	0.51
64 KiB	62453320	62512661	62503606	0.50
128 KiB	62461257	62509556	62510108	0.50
256 KiB	62300115	62510897	62500891	0.50
512 KiB	62567948	62508593	62500484	0.50
1 MiB	62572242	62553786	62492705	0.50
2 MiB	62577268	63477663	62492537	0.50

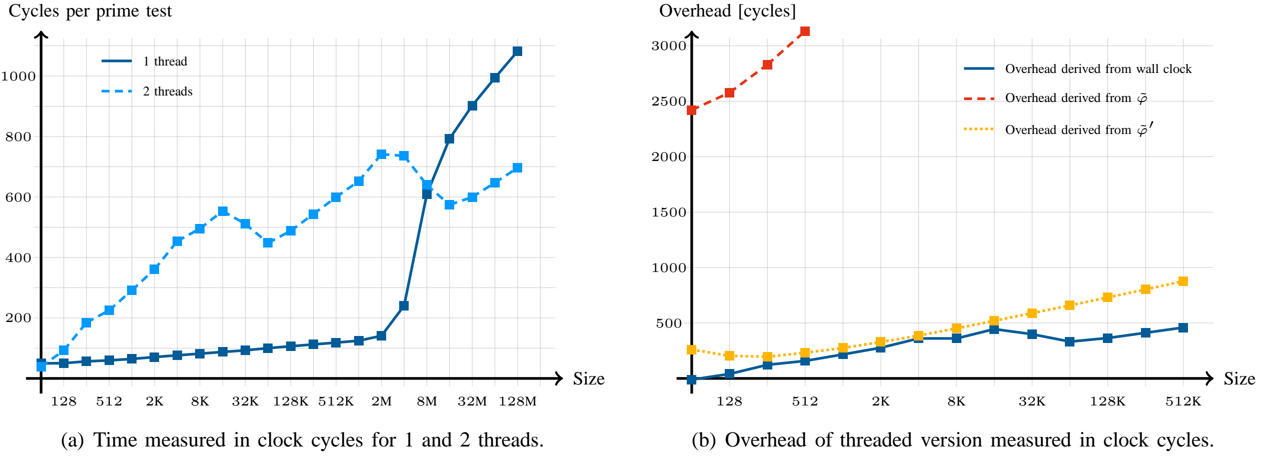
Table 1: Detailed results of the performance counters. For larger arrays the ratio is no longer meaningful since capacity misses occur.

around 125 cycles per memory reference and is about the same as for the store/store variant. However, our assumption is that this time only every second memory reference causes an FS event. Therefore, the penalty per FS event would be approximately the double and lie around 250 cycles. To proove this assumption we read again the performance counters using pfmon. The counter `L2_LINES_OUT:ANY` gives again the number of FS events assumed that no capacity misses occur. The total number of memory references is obtained by the `L1D_CACHE_LD:MESI` plus `L1D_CACHE_ST:MESI`. Again we can derive the ratio

$$\rho := \frac{\texttt{L2_LINES_OUT:ANY}}{\texttt{L1D_CACHE_LD:MESI} + \texttt{L1D_CACHE_ST:MESI}}.$$

between FS and memory references. The results are shown in Table 1. This time the value is considerable smaller than 1 except for array sizes between 256 and 1024 bytes. For sizes starting at 64KiB the ratio is exactly one half which means that an FS event occurs only every second memory reference. For larger arrays capacity misses occur which is why the delta of L2 evictions between the versions with and without FS have to be considered. By doing this the ratio drops to zero very quickly ($\rho = 0.07$ for 8MiB) which means that no more FS occurs. This proofs our assumption and gives a penalty per FS event of $\tau_c \approx 250$ cycles. By multiplying the ratios given in Table 1 with this value we end up with a close approximation of the overhead obtained by the wall clock approach. Consequently the penalty not only depends on the cores involved into an FS event but also on the operations performed. Our assumption is that a single store operation to a cache line can be buffered somewhere within a processor provided that this store does not depend on a value in the same cache line. This way the processors would be able to process the next store before the previous one has been written back in the store/store case. Modifying a value however causes a stall until the value has been loaded. Therefore, the FS penalty becomes completely visible. This result makes it even more difficult to choose the correct value for τ_c which now ranges anywhere between 50 and 250 clock cycles for our test system.

For comparison only Figure 6(b) shows the penalty per memory reference for cores with a common L2 cache. The average penalty adds up to 23 cycles per memory reference. Reading the performance counters gives similar results as for cores on different sockets wherefore the penalty per FS event again lies around 50 clock cycles.



(a) Time measured in clock cycles for 1 and 2 threads.

(b) Overhead of threaded version measured in clock cycles.

Fig. 7. Time and overhead per prime test.

```
(...)
int chunk = 1;
#pragma omp parallel shared(chunk) private(i,mask,tmp,id)
{
#pragma omp for schedule(static,chunk) nowait
   for( i=2; i<N; i++ ) {
      tmp = 2*i;
      while( tmp < max_size ) {
         mask = 128 >> ( tmp % 8 );
         ptr[ tmp/8 ] |= mask;
         tmp += i;
      }
   }
}
(...)
```

Listing 2. Parallelized loop of the prime sieve

B. Prime Sieve

The second test application is a prime sieve according to Eratosthenes. It is based on the idea to calculate all multiples of $n \in \{2, 3, \dots, N\}$ for a given upper bound N and to mark all numbers which are a multiple of another one. At the end all numbers which are not marked are prime. We implemented the most basic variant of this algorithm by using a bit array which represents the numbers to test. The single threaded version starts at 2 and marks all multiples up to N within the array by setting the corresponding positions to 1. Afterwards it starts over again with 3 and so on. At the end all bits representing non-prime numbers are set. The implementation consists of a single loop which can easily be parallelized using OpenMP (see Listing 2). We used a static parallelization to avoid any overhead induced by task assignment. The chunk size is set to 1 to ensure a tight interleaving between threads. This makes sense since otherwise OpenMP would divide the loop anywhere around $N/2$ which would lead to an unequal load balancing (there are fewer multiples smaller than N for a large number than for small numbers). In this case we pinned the threads to cores located on different sockets again. Figure 7(a) shows the average time per prime test measured in clock cycles for a given array size. Since the algorithm has no linear complexity the time per prime test increases for larger numbers. Parallelization fails for array sizes up to 8MiB which is double the size of the L2 cache. This is explained by the irregular access pattern of the algorithm. The stride sizes increase with the numbers tested and the latter part of the array is accessed more frequent than the former part, e.g. the first cache line is never accessed again after the first 512 numbers have been processed (64 byte times eight numbers per byte). The continuos line in Figure 7(b) shows the difference between the dual and single thread variant and therefore the overhead induced by parallelization. The dashed and dotted lines indicate the FS estimate $\tilde{\varphi}$ and $\tilde{\varphi}'$ as given by Pluto. To translate the estimates into a temporal context we used an estimated FS penalty of $\tau_c = 250$ cycles per event which is the highest average penalty for FS as obtained by the synthetic benchmark. This value was obtained by the synthetic benchmark where each thread modified the value in memory. Obviously $\tilde{\varphi}$ overestimates the penalty by more than an order of magnitude[2]. This is clear for two reasons. First, modifying values involves reading and writing and therefore a pattern of subsequent loads and stores for each thread. The estimate $\tilde{\varphi}$ assumes the worst case since it does not know the sequence of references. Second, the access pattern is not strictly sequential. The stride sizes vary which is why not every modify results in a FS event even for small array sizes. The result as given by $\tilde{\varphi}'$ is much more accurate. However, this result must be seen critical for the reasons mentioned in Section II-F. Furthermore,

[2]The slope of $\tilde{\varphi}$ is roughly linear for array sizes greater than 256KiB, not exponential as one could assume by looking at the section in Figure 7(b).

$\tilde{\varphi}'$ would also overestimate the penalty as the array size exceeds the L2 cache size since the cache size is not yet considered by Pluto.

V. Conclusion and future work

This report showed a new approach to detect and quantify false sharing effects. A formal model without temporal context was presented. Using this model a worst case estimate for false sharing and a best case estimate for true sharing were derived. These estimates were used to quantify false sharing. Since this approach does not rely on hardware support or precise simulation it is widely independent from hardware. It can also be extended to reflect distributed shared memory systems.
The most severe problem we encountered is the lack of temporal information about the sequence of memory references. Although the sequence of a single thread can be obtained by instrumentation the problem of deriving meaningful estimates in affordable time remains open. Without temporal information the estimate for false sharing turned out to be very inaccurate. An approac to sharpen the estimate is to take the possible number of true sharings into account. Although this led to good results in a first test we definitely have no longer and estimate of an upper bound for false sharing since it is easy to disprove the result by a counter example. Nevertheless this approach may turn out to be useful for a variety of real-world applications and should therefore not be discarded. Another possibility to sharpen the bounds may be to merge subsequent references to the same cache line. This reflects the abilities of most current processors and would dampen the estimates for some types of access patterns. Our original assumption that the cache size does not matter since false sharing would concentrate on a small number of frequently accessed memory blocks could not be affirmed so far. In contrast, the results for both examples turned out to be large overestimations for large working sets because capacity misses prevented false sharing in reality. Therefore, we plan to counter this problem by using the reuse distance [3] of memory blocks to dampen the estimates on large working sets which exceed the last level cache size. Further investigations regarding thread synchronization are necessary. Currently we support only barriers by making minor modifications to the source code of an application under test. Methods for automized detection of barriers is planned. Supporting partial synchronization between threads requires further investigation.
Finally the question remains open, if false sharing is really a problem for real-world applications. Although synthetic or deliberately bad written applications can show the possible effect of false sharing it is unknown how many real-world applications are affected. For lack of reliable mechanisms to detect false sharing, results of our approach are difficult to validate for all but self-written applications with simple access patterns.

VI. Documentation

This Chapter is meant as supplement for the comments contained in the source code to help others understanding the implementation faster. It is not a complete API reference but outlines the most important data structures and the ways in which they are accessed and modified. Please note that both tools are only meant to be a proof of concept and therefore in an experimental stage.

A. Pluto

Pluto is responsible to intercept any memory reference made by an application and to log statistical information. It is based on *Lackey*, another Valgrind tool which also collects statistical information about memory references. However, Pluto is interested in references to data on a more granular level than Lackey is. The reader is supposed to be basically familiar with Valgrind.

1) Instrumentation routine:

Any new basic block is passed to the instrumentation routine to allow Pluto to insert calls to its logging mechanism. We are only interested into memory reference to data, not instructions. This is based on the assumption that code is not modified at runtime and code is not stored within the same coherency block as data. This results only in load operations on the given memory blocks which is why FS can not occur in this case. Therefore, we do not insert calls to logging functions when a new instruction is loaded. The routine which decides if a call to the logging mechanism should be inserted is outlined in Listing 3. In case that the current statement `st` marks the beginning of an instruction we save the instruction address in a local variable. This is passed later on to the logging functions and finally used to obtain the code position of this instruction if debug information is available. In case of data load or store events calls to the corresponding logging functions are inserted. The instruction address is passed on to these functions together with the size of the memory reference and its address. This address can be used to obtain the start address of a memory block and the offset of the memory reference. The thread ID is not known yet. It is determined by the logging function on the fly.

2) Data structure for statistics:

To store statistics about each coherency block the logging functions use a two level hash map as depicted in Figure 8. An instance of `struct CBlock` represents a coherency block within a single parallel section. The key used to address this element within the hash map is the start address of this coherency block within memory. The size of `UWord` depends on the platform (32/64bit). Each instance of `struct CBlock` contains another hash table where information about access classes

```
(...)
switch (st->tag) {
   case Ist_IMark:
      addStmtToIRSB( sbOut, st );
      ins_addr = mkIRExpr_HWord( (HWord)st->Ist.IMark.addr );
      break;

   case Ist_WrTmp:
      addStmtToIRSB( sbOut, st );
      data = st->Ist.WrTmp.data;
      if (data->tag == Iex_Load) {
         addEvent_Dr( sbOut, data->Iex.Load.addr, sizeofIRType(data->Iex.Load.ty), ins_addr );
      }
      break;

   case Ist_Store:
      data = st->Ist.Store.data;
      addEvent_Dw( sbOut, st->Ist.Store.addr, sizeofIRType(typeOfIRExpr(tyenv, data)), ins_addr );
      addStmtToIRSB( sbOut, st );
      break;

   case Ist_Dirty:
      di = st->Ist.Dirty.details;
      if( di->mFx != Ifx_None ) {
      tl_assert(di->mAddr != NULL);
         tl_assert(di->mSize != 0);
         if( di->mFx == Ifx_Read || di->mFx == Ifx_Modify || di->mFx == Ifx_Write) {
         tl_assert(0);
      }
      addStmtToIRSB( sbOut, st );
      break;

   case Ist_Exit:
      flushEvents(sbOut);
      addStmtToIRSB( sbOut, st );
      break;

   (...)
}
(...)
```

Listing 3.   Instrumentation routine

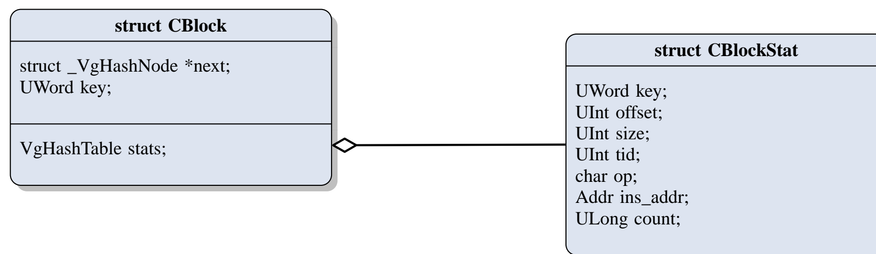


Fig. 8.   Hierarchy of hash maps used to log information about memory references

to this block are stored. An access class is identified by a key derived from a string containing the instruction address which contains

- the instruction address which caused the memory reference,
- the size and offset of the memory reference,
- the thread which executed the instruction
- and the type of memory reference (load or store)

in this order. The string is hashed by Robert Sedgwick's (Algorithms in C) hashing function [1]. Other hash functions are ready to use if necessary. The key size is currently 32bit for all systems but can easily be extended to 64bit. The elements of this second hash table are instances if `struct CBlockStat` which essentially stores the number of references of this type as well as the attributes mentioned above.

3) Logging mechanism:

The logging mechanisms for load and store operations work in the same way:

1) Calculate start address of the affected coherency block (key for the `CBlock` hash map) and the offset of this access.

2) Decide if the access spans one or two coherency blocks. [3] If the access spans two coherency blocks it is split into two different accesses each covering only one block.
3) Lookup the affected block in the `CBlock` map and create it if it does not exist yet.
4) Determine the ID of the thread which is currently running.
5) Create the hash key for the statistics map of this block.
6) Search for a similar memory reference in the statistics map. If there is no entry, create a new instance of `CBlockStat` and insert it into the statistics map.
7) Increment the counter for this class of references.
8) Repeat these steps 3) to 7) for a second block if the access spans more than one block.

4) Considering barriers:
Barriers are currently not recognized automatically. Instead Pluto relies on client requests to be notified about barriers. These client requests are inserted into the application under test before and after a parallel section. When a thread executes the client request it tells Pluto its thread ID and if it encountered or passed a barrier. Additionally Pluto keeps track of thread creation and therefore knows the number and IDs of all threads currently running. Using this information Pluto can determine if all threads have reached a barrier as follows:

1) When a thread executes a client requests and tells Pluto that it reached a barrier, Pluto sets a flag for this thread.
2) Every time a thread reaches a barrier, Pluto checks if all threads currently running reached this barrier by checking their flags. If all threads reached the barrier Pluto initiates a flush of its logging buffers to write the statistics for this parallel section to file. Afterwards Pluto clears its hash tables and prepares for a new parallel sections.
3) When a thread executes a client request and tells Pluto that it passed a barrier Pluto immediately checks if the barrier flags of all threads are set. If this is the case, the thread currently passing the barrier is the first one which executes code after the barrier. In this case Pluto clears the barrier flags of all threads. If the barrier flag for at least one thread was not set, the thread currently passing the barrier did this before another thread reached the barrier. Therefore, Pluto knows that partial synchronization between threads happened and prints a warning about this event.

The mechanism described above is implemented by means of two bit masks supporting a total of 63 threads (thread ID 0 is reserved). The masks are initialized with zero. For each thread created the bit number corresponding to its thread ID is set to 1. If a barrier is reached, the bit position of the thread which reached the barrier is set to one. Therefore, the checks reduce to simple logical operations performed on these two bit masks.

5) Flushing the log buffers:
The log buffers are flushed if a new parallel section begins or the guest application terminated. In both cases Pluto iterates over the coherency blocks and all statistical entries within this block. The statistics per block are sorted by thread ID which simplifies the work of the parser later on. While iterating over the statistical entries Pluto determines if this block has been accessed by at least two threads. Only if this is true, the block is considered interesting and written to file. At this time the address of the instruction which caused the reference is considered again and translated to a code position (file name and line number) if possible. After the information has been written to file the hash maps are destroyed. Note that filtering memory blocks at this point disallows for considering sharing events between parallel sections as described in II-I. Skipping this check and writing all data to file is possible without requiring modifications of the parser. Adapting the parser is only necessary if it must support sharing events between parallel sections.

6) Output file format:
The output file is plaintext and is organized as follows:

```
(...)
na,1,37,1,l,1
na,1,37,1,l,1
====
[6299648]
na,0,32,8,s,1
na,0,0,8,s,1
na,1,32,8,l,2
[6299776]
eratosthenes.c:78,0,40,8,l,2759
eratosthenes.c:78,1,40,8,l,2447
(...)
```

Listing 4. Example for the output format of Pluto

The delimiter `====` is used to separate parallel sections. A new coherency block is indicated by its address in decimal form enclosed by brackets. Afterwards each statement describes a class of memory references in the order code position, thread ID, offset, access size, type and count. If no debug information have been available at runtime the code position is set to `na`.

[3]Note that in case of 64bit systems a single memory reference can never span more than two cache lines. This is why we currently only support 64bit systems. In case of 32bit systems a memory reference may be larger and therefore need special handling.

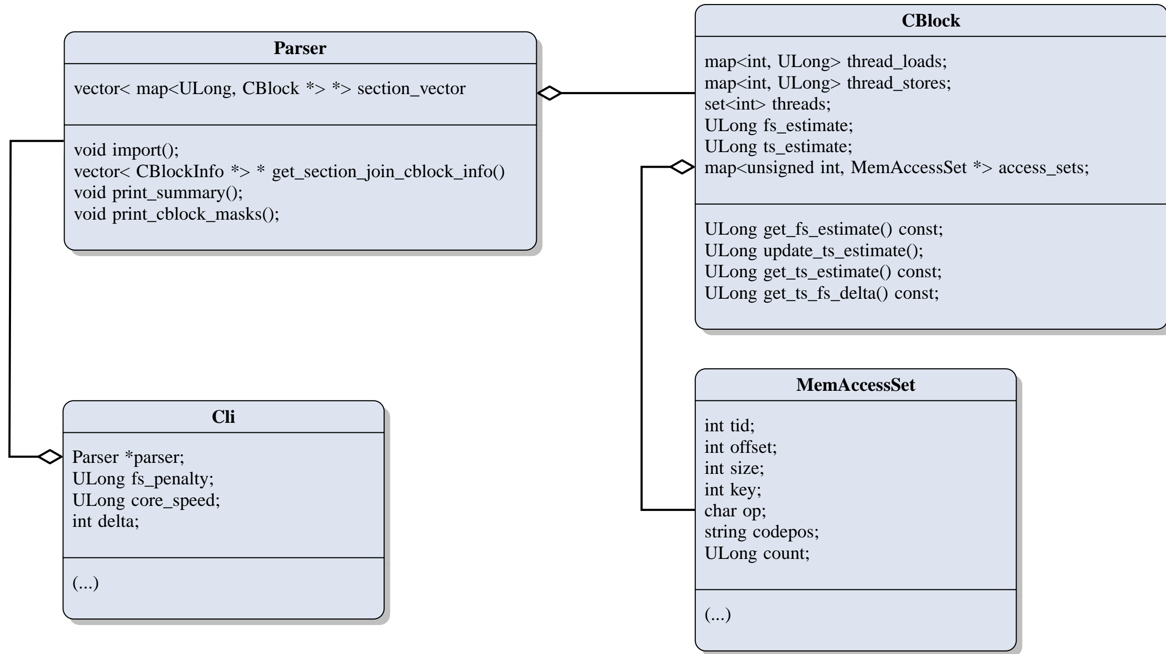


Fig. 9. Class hierarchy of the parser

B. Output parser

The parser is responsible to read the log file of Pluto and to calculate the estimates. Basically this could be done by Pluto. However, doing it within Pluto has two drawbacks. First, it would require to re-run Pluto each time an option changes, e.g. the penalty per FS event or the core speed. Second, the parser heavily relies on hash maps, vectors and sets as provided by the C++ Standard Template Library (STL). For these reasons we decided to use an external parser. The parser reads the log file and builds a data structure similar to the two-level hash map used by Pluto. Afterwards it shows a small command line and waits for further commands. After receiving a command it iterates over this data structure and collects the information necessary to calculate the estimates for TS and FS.

1) Command line reference:

The parser is started by supplying the logfile to analyze as single argument. Afterwards the parser automatically imports the file prompts for further commands. Possible commands are:

- `set corespeed <value>`
  Specify the core speed in MHz. This value is needed to convert the estimates to an overhead in milliseconds. If this value is not given, the parser omits the temporal overhead and prints only the estimates.
- `set penalty <value>`
  Specify the penalty per FS event in clock cycles. The default value is 50.
- `set delta <value>`
  Specify if the temporal estimate should be calculated by using $\tilde{\varphi}$ or $\tilde{\varphi}'$. The default is 0 which corresponds to $\tilde{\varphi}$. Set it to 1 to use $\tilde{\varphi}'$. Output of multiple coherency blocks is ordered according to the active estimate.
- `show summary <value>`
  Print a list of coherency blocks ordered by either $\tilde{\varphi}$ or $\tilde{\varphi}'$ dependent on the current value for delta.
- `show cblock <address>`
  Print details about a specific coherency blocks, including its access masks and code positons of references. The address must be given as hex value with leading `0x` as given by the summary.

2) Class hierarchy and data structure:

The most relevant section of the class hierarchy is shown in Figure 9. Note that the class description is limited to the most relevant members. An instance of class `Parser` is created by the main method. Afterwards the parser imports the log file. For each parallel section a new hash map is created which will store the coherency blocks. The hash maps are organized in a vector whose length corresponds to the number of parallel sections encountered during import. The section maps contain pointers to instances of class `CBlock`. Within this class most of the logic is concentrated. The variables `thread_loads` and `thread_stores` correspond to $L^{(t)}$ and $S^{(t)}$ and are used to calculate $\tilde{\varphi}$ which is stored in `fs_estimate`. Basically these variables are shortcuts which are created and updated as the log file is parsed. All information could also be derived when it is

needed from the access classes. These are stored in the hash map `access_sets` which corresponds to $\{L, S\}_{x,y}^{(t)}$ for the given block. The associative container `threads` contains the thread IDs of all threads which accessed this coherency block and is used to iterate over tuples of threads during calculation of the estimates. After the import is complete an instance of class `Cli` is created to which the parser instance is connected. Afterwards control is passed to the command line interface which waits for user input. At this point all coherency blocks within each section yield valid entries for $\tilde{\varphi}$ and $\tilde{\theta}$. However, the information is still spread over multiple sections. Listing the coherency blocks triggers the `print_summary()` member method of the parser. This in turn triggers a sequence of actions to merge the information of all sections into temporary variables for each coherency block called `CBlockInfo`. This is done to preserve the information of the original instances for further analyzes. Afterwards the temporary vector of info elements is sorted by $\tilde{\varphi}$ or $\tilde{\varphi}'$ and printed on screen. In a similar way investigating a specific coherency block triggers the `print_cblock_masks()` member which collects the necessary information for the given block over all sections, creates the access masks and prints the output. At any time the section vector remains a read only copy of data and should never be modified.

REFERENCES

[1] Arash Partow. General Purpose Hash Function Algorithms Library, 2002.

[2] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[3] Xiong Fu, Yu Zhang, and Yiyun Chen. Data-layout optimization using reuse distance distribution. In *Emerging Directions in Embedded and Ubiquitous Computing*, pages 858–867. Springer Berlin / Heidelberg, 2006.

[4] Intel Corporation. Intel Performance Tuning Utility 3.2, User Guide, chapter 7.4.6.5, 2008.

[5] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.