Cache Influence on Worst Case Execution Time of Network Stacks

Jork Löser Hermann Härtig

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: jork.loeser@inf.tu-dresden.de

Abstract

We apply the cache partitioning technique on a network stack to derive the influence of caches on worst case execution times of complex applications. We demonstrate that the overhead caused by cache misses when receiving packets of typical sizes from the network is reduced from 310% to 90% compared to the best case. For the transmit direction, cache partitioning results in a reduction of the overhead from 78% to 26%.

1 General

This paper leads together two, so far independent lines of research, namely cache partitioning for real-time systems and real-time networks. Both are related to worst case analysis of system components, but have never been looked at in combination.

In real-time systems, admission of new tasks or connections depends on their worst case resource requirements, e.g. on their worst case execution time.

In architectures with caches, a significant point to look at for the analysis of real worst-case execution times is the cost for memory accesses. A well studied approach for bounding the worst case is cache partitioning (e.g. [LHH97, Mue95, Wol93]), a technique to partition the cache among applications and thus to isolate applications with respect to their cache usage. In [LHH97], the authors provide measurements for simple applications such as a matrix multiplication and digital filters which were highly tuned to make optimal use of caches and hence to suffer as much as possible from cache misses. However, the influence on complex real applications and applications which are not tuned for optimal cache usage was not examined and left to future experiments.

In [BH98], design, implementation and evaluation of an ATM-based network stack is described, that provides quality-of-service (QoS) -guarantees to applications even in the end systems, which are otherwise oftenly neglected. This is achieved by reservation, accounting and policing on a per-connection basis. Worst case execution times of the operations in the network stack were measured. Reservation is based on these execution times, on the buffer requirements and on the bandwidth requested.

The requirement for policing of non-conforming connections places a heavy worst case load on end systems, especially in the receive direction. However, the authors did mention, but not analyse the influence of caches; hence the effect of caches on worst case execution times of the network stack is unknown.

In this paper, we apply cache partitioning techniques as described on [LHH97] to analyse the effect of caches on the network stack described in [BH98]. This contributes to a better understanding of cache influences on more complex components than filter and matrix multipliers, and it contributes to a better understanding of real worst case resource consumption in a network stack.

The remainder of the paper is organized as follows: In Section 2 the technique of cache partitioning and the network stack are described. Section 3

introduces the environment used for our measurements and describes the applications forming the system. The cache flooding algorithm to achieve worst case execution times on further memory accesses is introduced here. In Section 4 the results obtained from measurements are presented and discussed. Finally, some conclusions are presented.

2 Background

This section introduces the topics we relate to in this paper in more detail.

2.1 Cache Partitioning

Ideal cache partitioning limits the cache access pattern of one application to the cache partitions the application is mapped to. No influencing of other applications caused by cache interferences occurs. As a result, memory access times will mainly be determined by the application itself.

The cache partitioning technique described in [LHH97] uses the virtual to physical address translation of modern computer architectures to partition physically-tagged caches. Its major advantage compared to other cache partitioning techniques is the transparency to applications not relying on the physical addresses of a virtual memory address. The technique is limited to a minimum cache size given by the multiple of the physical page size and the associativity of the cache. In a 2-level cache hierarchy this typically restricts this technique to partitioning the L2 cache. The L1 is shared by all applications, so that we expect measurable differences between the best case and the the worst case with cache partitioning and flooding the cache by parallel applications. Other sources for additional overhead not influenced by cache partitioning include misses at the translation look-aside buffer (TLB). They are discussed in detail in [LHH97].

2.2 Network Stack

The network stack we have used is part of the Dresden Realtime Operating System (DROPS, [HBB+98]). Using a modified socket interface, ATM adaption layer 5 (AAL 5) is supported. It consists of 2 servers, each running in its own address space. The first server is the driver for the ATM network card, the second server is responsible for QoS reservation and enforcement. Communication between the servers and the client applications is done by the L4 inter-process communication mechanism (IPC).

The ATM driver uses a synchronous interface in transmit direction, thus a send function returns after the data of a AAL5 protocol data unit (PDU) is transfered into the internal buffers of the ATM card. So the application buffers can be reused immediately after returning from the send request. In the receive path, the driver operates in interrupt driven mode. The ATM card copies the data into communication buffers in main memory and sends one interrupt per copied AAL5 PDU to the processor. The driver copies the data to its client to free these communication buffers.

The QoS server is the only client of the ATM driver. Both run multithreaded, so receive and transmit can occur simultaneously. A transmit operation from a client results in sending a message to the QoS server. Then the server checks the account of the client and sends a transmit-request to the ATM driver. The driver signals the address, length and other information to the network interface and waits for completion. Because of the synchronous interface, the data need not to be copied inside main memory, resulting in a zero-copy architecture. At the receive path, data is copied once in the ATM driver, and a second time inside the QoS server. After checking the receive-account of the associated client the QoS server signals the reception the the client. For data transfer between QoS server and client a shared memory region is used. To sum it up, at the receive path the data is copied twice, at the transmit path we have a zero-copy architecture.

3 Measurement Setup

In this section we introduce the concrete hardware and software architecture and describe the components used for the measurements.

3.1 Structural Setup

The environment used for our experiments is based on the L4 μ-kernel [Lie95]. It supports multiple address spaces and allows physical memory to be

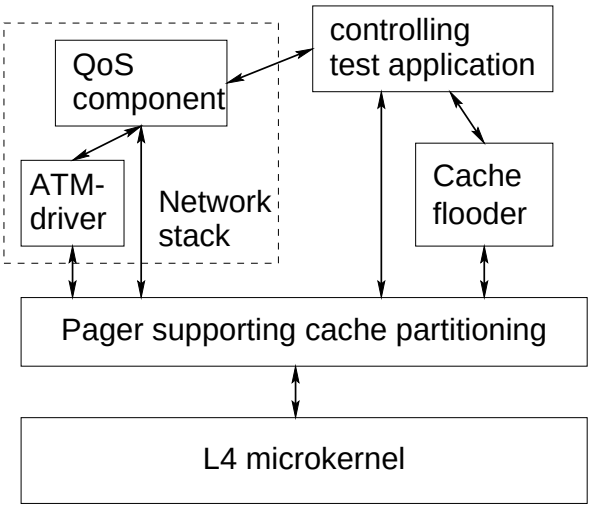

Figure 1: The components of the system

managed by user tasks. Based on that, a user-level pager was written supporting cache partitioning for the applications.

The applications running on the system are the pager, the two servers of the network stack, a test application and a cache flooder, see Figure 1.

The cache flooder is responsible for achieving the worst case memory cache situation, i.e. the execution time of further memory accesses is maximized. The algorithm is described in detail in Section 3.2. It runs in its own address space which allows to be restricted by cache partitioning as needed by the experiments. The flooding can be triggered at any time by explicit request.

The test application controls the experiments and measures the time. To obtain the effect of cache partitioning to worst case execution times of the network stack, data packets were send through the stack in both directions. Interruptions of operations at stack are suppressed, their influence is derived based on the measurement results later. Four measurements were made for each direction, with cache flooding and without cache flooding and with cache partitioning enabled and disabled, in any combination. When cache partitioning was enabled, it was ensured that each component had its own cache colors respective its own L2 cachelines. As discussed in Section 3.2.2, the cache flooder reserved an additional cache partition, independent if the applications share the remaining cache or not.

Cache was flooded differently for transmit and receive. When cache flooding was used in transmit direction, the flooder was triggered immediately before the send operation. The time measured for transmit direction includes the operations at client for sending the transmit-request to the QoS server, the actions taken in the QoS Server and the ATM driver, the time for sending the data to the ATM card and all the signalling back to the client.

The ATM network interface uses internal buffering on the network card. Due to interleaving effects this results in shorter transmit times for small PDUs, which must be considered in the QoS component. To eliminate the delays caused by occupied buffers, we insert gaps between the send operations.

The receive operation consists of two separate operations in worst case. The first operation covers sending a receive-request to the QoS server and all preparations for the next incoming PDU. Then the network stack is waiting for the interrupt (IRQ) signalling the received PDU at the ATM card. The second operation covers the action taken in reaction to the incoming IRQ. In the ATM driver the PDU is pushed to the QoS server, the QoS server does several operations and signals the PDU to the application. When cache flooding was used on receive direction, the flooder was triggered twice per receive request. The cache was flooded once before the receive-request and again after all preparations for the next incoming PDU were done. The time measured for receive direction includes the actions taken in the the client for sending the receive-request, the actions taken in the QoS Server and the ATM driver until the wait for the IRQ. The measured time also includes the actions taken in reaction to the incoming IRQ, signalling the PDU acceptance to the ATM card, copying the data to the QoS server and copying the data from the QoS server to the client.

3.2 Flooding the Cache

In this section we describe the cache flooder used for achieving worst case execution times in our experiments. We introduce the worst case situation for memory accesses and present algorithms for achieving this situation on different cache architectures.

Due to the dependency of memory access times

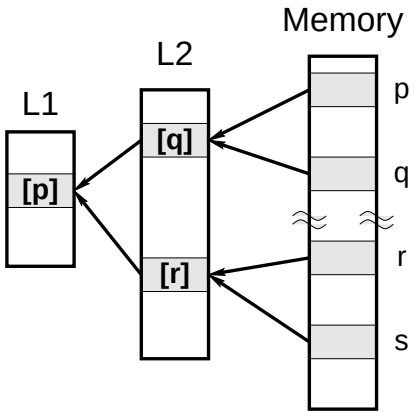

Figure 2: Worst-Case scenario for memory access to s. Arrows denote the mapping relations from memory to L2 and from L2 to L1. Bold entries denote modified data.

on the underlying architecture, we need to describe our assumptions on the system. We assume a 2-level memory cache architecture as common on modern processors. The caches are tagged by physical addresses and support a write back strategy. For memory write accesses from the processor, write allocation is used. No write allocation is used in L2 when when writing back a modified line from L1, if this would require to write back a line from L2 first. Caches are set-associative and use a strict LRU cacheline substitution mechanism, n-way means a cache with associativity n. A 1-way cache is called direct mapped. Further we assume L2 to be larger than L1; the concrete limit, where common architectures are within, is derived in Section 3.2.2. When showing configurations of caches with multiple associativity, the least recently used element will be written to the left and the last accessed element to the right. At the next access to a new element, the leftmost will be purged.

The granularity of reading and writing data in caches is a cacheline. Therefore we always use cacheline numbers as addresses, the bits determining a byte inside a cacheline are neglected. The same applies for memory addresses. With [x] we denote the contents of a cache element belonging to the (reduced) memory address x.

3.2.1 Worst-case on Memory access

The worst case on a memory access corresponds an access to memory address s in Figure 2. [s] can neither be found in L1 nor in L2, hence it must be loaded into both first. The L2-line s is mapped to contains modified line r. The L1-line s is mapped to contains modified line p. To store line s in L1, a write back of p must be performed to free the cache entry. The L2-line p is is mapped to contains another modified line, q. This results in a write-through of line p to the main memory. A write-back must be performed on the L2-line containing r too, because the entry contains a modified line. Then, the line containing s can be loaded into L2, L1 and the processor. To sum it up, the one memory reference results in 2 write accesses and 1 read accesses to main memory. We refer to this as double purge case. With double purge configuration we denote the cache configuration leading to the double purge case at the next memory access.

3.2.2 Achieving the Worst Case

This section derives algorithms to achieve the double purge configuration described in the previous section. We look at different cache structures and construct the double purge configuration for each.

Construction of the double purge configuration requires that the inclusion property [BW88] is violated, i.e. L1 contains cachelines that are not in L2. Whether or not the inclusion property is guaranteed, is determined by the cacheline substitution algorithm [BW88, WBL89]. The LRU algorithm does not ensure the inclusion property, so the double purge configuration can be constructed. How this can be achieved depends on the hardware structure of the caches, to be precise, on their associativity.

Because of the cache hierarchy, every line being loaded into L1 is also loaded into L2 (and may be substituted later). As a result, the double-purge case requires at least a 2-way L1. Otherwise all lines in L1 also reside in L2.

Prior to discussing the algorithms for flooding the cache, we point out some predicates and introduce additional notations. A 2^n-way cache of size 2^m has an address width of $i = m - n$ bits. To index an element in this cache, only the least significant i bits are used. With a reduced memory address width

step	access	effect	comments
1	wrt A0.2.0	L1[0] := [A0.2.0]	load the line the double purge case will be achieved for
2	wrt A0.0.0	L1[0] := [A0.2.0], [A0.0.0]	load line 0 into L1 and L2
3	wrt A1.0.0	L2[2.0] := L1[0]:[A0.2.0] L1[0] := [A0.0.0], [A1.0.0]	purge [A0.2.0] from L1 to L2
4	wrt A0.1.0	L2[0.0] := L1[0]:[A0.0.0] L1[0] := [A1.0.0], [A0.1.0]	L2[0.0] is filled with modified line, disjunct to L1

Table 1: Access pattern for achieving the double purge configuration on a direct mapped L2 for addresses Ax.2.0, x>0.

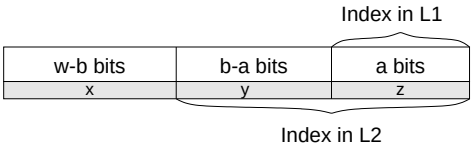

Figure 3: Splitting the address into cache representatives

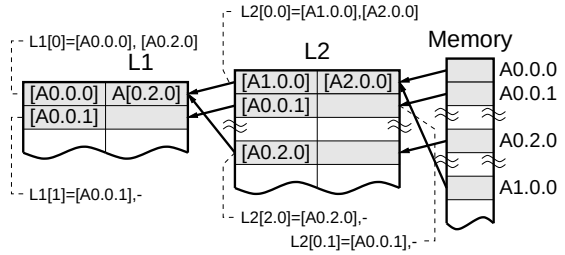

Figure 4: Illustration of notations and mappings from memory to L2 and L1. Unknown/empty elements of caches are denoted with '-' in the entry-lists.

of w bits, 2^{w-i} elements of memory are mapped to every element in the cache. In a 2-level cache hierarchy with an L1 address width a and an L2 address width b, 2^{b-a} elements of L2 are mapped to every element in L1. Because we use this to purge specific lines from L1 and L2, we assume L2 to be as least $a * b$ times bigger than L1. This is true for almost all architectures. With Ax.y.z we denote a memory address with z holding the least significant a bits of the address, y holding the next b bits and x holding the most significant $w-b$ bits. Addresses with identical values for z are mapped to the same L1 element. Analogously, addresses with identical

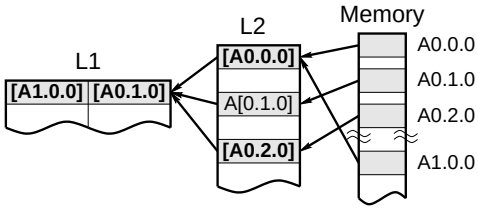

Figure 5: Worst case on a 2-way L1 and a 1-way L2: access to addresses Ax.2.0, x>0. Bold entries denote modified lines.

values for y and identical values for z are mapped to the same L2 element. So x specifies which representative of the memory-L2 mapping will be addressed in L2, and y specifies the representative of the L2-L1 mapping in L1. We denote the list of entries at index z in L1 with L1[z]. Its length is a, the associativity of L1. With L2[y.z] we denote the list of entries at index $y.z = y * 2^a + z$ in L2, having a length of b. See Figure 4 for an illustration of terms and addresses.

Direct Mapped L2

We construct the worst case configuration for a 2-way L1 cache and a direct mapped L2 cache. Firstly, the double purge configuration is achieved for a single L2-line and the corresponding memory entries. See Table 1 for the access patterns and Figure 5 for the result.

Writing in the first step to lines A0.y.0 with y in $[2..2^{b-a})$[1] brings all L2-addresses of these lines into worst case state. To achieve the double purge case

[1] '[' denotes including, ')' excluding the given bound.

step	access	effect	comments
1	wrt A0.2.0	L1[0] := [A0.2.0], -	load the line the double purge configuration will be achieved for
2	wrt A1.2.0	L1[0] := [A0.2.0], [A1.2.0]	load another line with the same L2 address
3	wrt A0.0.0	L2[2.0] := L1[0]:[A0.2.0], -	purge [A0.2.0] from L1 to L2
		L1[0] := [A1.2.0], [A0.0.0]	load line 0 into L1
		L2[0.0] := [A0.0.0], -	and L2
4	wrt A1.0.0	L2[2.0] := [A0.2.0], L1[0]:[A1.2.0]	purge [A1.2.0] from L1 to L2 L2[2.0] contains two modified lines
		L1[0] := [A0.0.0], [A1.0.0]	load another line into L1
		L2[0.0] := [A0.0.0], [A1.0.0]	the line is also loaded into L2[0.0]
5	wrt A0.0.0	L1[0] := [A1.0.0], [A0.0.0]	make [A0.0.0] the least recently used element in L1, but not in L2
6	wrt A2.0.0	L2[0.0] := [A0.0.0], L1[0]:[A1.0.0]	purge [A1.0.0] from L1 to L2
		L1[0] := [A0.0.0], [A2.0.0]	load another line into L1 and L2.
		L2[0.0] := [A1.0.0], [A2.0.0]	This purges [A0.0.0] from L2, inclusion property is violated.
7	wrt A0.0.0	L1[0] := [A2.0.0], [A0.0.0]	make [A0.0.0] the least recently used element in L1, but not in L2
8	wrt A0.1.0	L2[0.0] := [A1.0.0], L1[0]:[A2.0.0]	purge [A2.0.0] from L1 to L2
		L1[0] := [A0.0.0], [A0.1.0]	load another line into L1 and L2
		L2[1.0] := [A0.1.0], -	

Table 2: Access pattern for achieving worst case situation on a 2-way L2 for addresses Ax.2.0, x>1.

for almost all memory accesses, the whole flooding operation must be repeated for all L1-lines, i.e. the least significant bits set to 1, $2 \ldots 2^a - 1$. The double purge case will occur for all addresses Ax.y.z with y>1.

2-way L2 Cache

Not considering L1, the following algorithm floods all L2-lines with address L2[0.0]:

```
for( x = 0; x < ass(L2); x++)
  wrt(Ax.0.0);
```

For flooding L1 with lines not contained in L2, we select lines A0.0.z, to stay in L1 and lines A1.0.z and A2.0.z to stay in L2. We touch the lines to stay in L2, and in between we touch the lines that will stay in L1. Given an n-way L1, the LRU algorithm executed by the caches allows pinning of n-1 modified lines in L1 this way. The algorithm in Table 2 constructs the worst double purge configuration for selected lines of a 2-way L1 and a 2-way L2. The resulting configuration is shown in Figure 6.

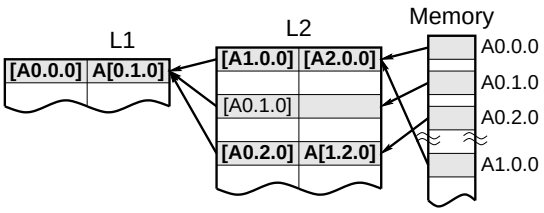

Figure 6: Worst case on a 2-way L1 and a 2-way L2: access to addresses Ax.2.0, x>1. Bold entries denote modified lines.

Accessing any of the lines Ax.2.0, x>1 results in writing back the modified line A0.0.0 from L1 to memory and writing back the modified line A0.2.0 from L2 to memory (see Figure 6). To enhance the algorithm to almost all addresses, execute

```
for( y = 2; y < 2^(b−a); y++)
  for( z = 0; z < 2^a; z++)
      { wrt(A0.y.z); wrt(A1.y.z); }
for( z = 0; z < 2^a; z++)
  { wrt(A0.0.z); wrt(A1.0.z);
  wrt(A0.0.z); wrt(A2.0.z);
  wrt(A0.0.z); wrt(A0.1.z); }
```

This achieves the worst-case situation for addresses Ax.y.z with x>1 and y>1.

Access to the addresses Ax.1.z do not meet the double purge case criterion, even not to addresses A0.y.z and A1.y.z. The latter is clear, because these lines already reside in L2 due to the first flooding loop. Access to the lines Ax.1.z would not require to write back a L2-line, because one of the L2-entries contains an invalid line, which can be purged. The corresponding modified entry resides in L1. In Figure 6 this is line [A0.1.0]. As a result, access to the these lines must be circumvented when measuring worst-case execution times of applications. This can be achieved by cache partitioning: one partition of L2 will be reserved for the cache flooder to hold the addresses Ax.1.z. The second demand, not to share the addresses A0.y.z and A1.y.z with other applications is done by usual memory management.

General Case

To generalize the cache flooding algorithm, we assume an n_1-way L1 and a n_2-way L2, $n_1 > 1$. In the first step, we touch most of the L2 by executing

```
for( x = 0; x < n_2; x++)
 for( y = 2; y < 2^(b−a); y++)
     for( z = 0; z < 2^a; z++)
         wrt(Ax.y.z);
```

In the second step we execute

```
for( z = 0; z < 2^a; z++)
  {
  for( x = 0; x < n_2; x++)
      {
      for( j = 0; j < n_1 − 1; j++) wrt(Aj.0.z);
      wrt(A(x+n_1 − 1).0.z);
      }
  for( j = 0; j < n_1 − 1; j++) wrt(Aj.0.z);
  wrt(A0.1.z);
  }
```

With these access patterns we achieve the worst-case situation for addresses Ax.y.z with x>1 and y>1.

Again, cache partitioning is needed to systematically achieve the worst case in memory access times with general applications. Also, the memory used for flooding must not be shared with other applications to get the worst case.

3.3 Hardware Architecture

The architecture we used for our experiments was an Intel Pentium-II-based PC. The Pentium-II disposes of a 16KB 4-way L1 data cache (D-cache), a 16KB 4-way L1 instruction cache (I-cache) and a 512KB 4-way unified L2 cache. Both the D-Cache and the L2 cache are write back caches and support write allocation for write-access from processor to main memory.

Because the I-cache does not support write-back, the worst case for instructions is not to find the corresponding code in the L1. For the L2-part, the worst case corresponds to a modified line which has to be written back first. This can easily be achieved by flooding L1 I-cache by executing a sufficiently huge chunk of code and using the cache flooding technique described above for flooding the whole L2 cache and the L1 D-cache.

The general cache flooding algorithm described in Section 3.2.2 was adapted and used by the cache flooder application.

4 Measurement Results

When cache flooding was not used, nearly no differences were measured between the partitioned and the unpartitioned cases. So we do not distinguish between these two cases when discussing the results.

4.1 Transmit Path

Figure 7 shows the number of CPU cycles needed for the transmit path of AAL5-PDUs of different sizes. The lower graph corresponds the unflooded case, i.e., only send operations with sufficient breaks in between are executed. Because the cache is not modified by other instances, this is assumed to be the best case. The upper graph shows

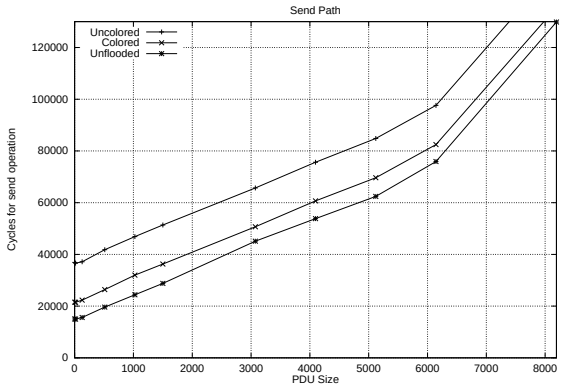

Figure 7: Cycles needed for sending AAL5-PDUs of different sizes

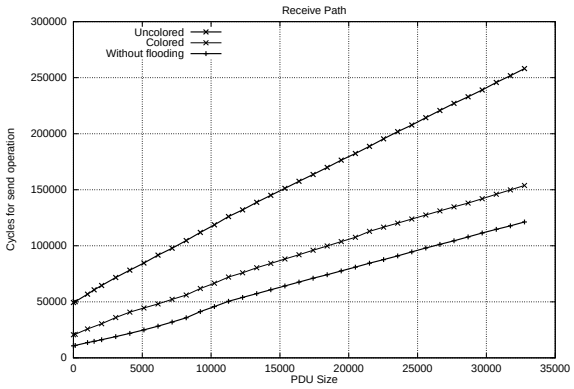

Figure 8: Cycles needed for receiving AAL5-PDUs of different sizes

the cycles needed when all applications share the whole cache and the cache flooder is able to flood the cache of all these applications. Assuming the network stack is uninterruptible, the upper graph corresponds to the WCET. The graph in the middle shows the colored case, where each component has its own cache partition. The cache flooder was limited to its own part of the L2 cache, but still influences the L1 cache and the TLB, as mentioned in Section 2.1. For the smallest possible PDU size, cache partitioning reduces the maximum overhead caused by cache misses in the send path from 145% to 43% compared to the best case. For PDU sizes of 4KB the maximum overhead is reduced from 40% to 12%. For PDU sizes equal to the typical maximum transfer unit (MTU) in Ethernet networks of 1500 Bytes, the WCET is reduced from circa 51000 cycles to 36000 cycles by cache partitioning. This tightens the range from best case to worst case from 78% to 26%, which results in a more accurate resource reservation.

As already stated, the ATM network interface uses internal send buffers. They are responsible for the bend at PDU sizes of about 6KB.

4.2 Receive Path

Figure 8 shows the number of CPU cycles needed for the receive path of AAL5-PDUs of different sizes. The execution time is calculated as the sum of the both parts of the measurements. The upper graph corresponds the uncolored case, the cache was flooded twice per receive operation. This is assumed to be the WCET. The lower graph shows the results from measurements without flooding the cache. We assume it to correspond the best case in the receive path. The graph in the middle shows the results obtained from flooding in combination with cache partitioning. This corresponds to the worst case when cache partitioning is used. The cache flooder was limited to its own part of the L2 cache, but still influences the L1 cache and the TLB. The effect of cache partitioning for 1-byte PDUs is a reduction of the overhead caused by cache misses from 360% to 92% compared to the best case. For a PDU size of 4KB the overhead is reduced from 260% to 86%. For PDU sizes equal to the typical MTU in Ethernet networks of 1500 Bytes, the WCET is reduced from circa 60700 cycles to 28000 cycles by cache partitioning. This tightens the range from best case to worst case from 310% to 90%.

4.3 Discussion of the Results

The receive path profits more from cache partitioning than the send path. One reason for this is the double copying of every data which is received, while the send path requires no copy. Hence, the influence of cache misses in the send path are independent of the PDU size, conflicts occur only at the control path. For receive direction, cache misses also occur for the data. The consequence is a dependency of the cache hit rate from the PDU

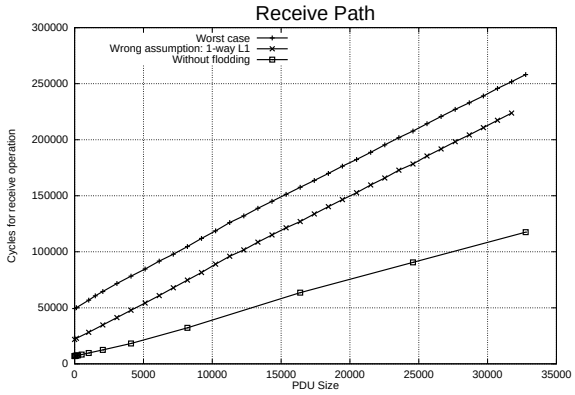

Figure 9: Missing the WCET by wrong assumption of associativity.

size. Up to a certain limit, based on the cache size, this dependency is linear. Another reason for the stronger influence to the receive path is the two-step mode of operation. So the cache can be flooded twice per receive procedure. While cache partitioning reduces the overhead in transmit path by 15,000 cycles, the overhead in the receive path is reduced by ca. 30,000 cycles for the smallest PDU size.

Generally, exact measurements of WCET are a nontrivial task in complex systems like the x86 PC architecture. The construction of the cache flooder emphasized that comprehensive knowledge of the cache architecture is needed. If the wrong associativity or the wrong cache sizes are assumed, the worst case scenario is not achieved. For example, flooding the cache hierarchy by just writing into a large array is appropriate for direct mapped L1 caches. With the 4-way L1 cache, this leads to execution times which are 11,000 cycles below the WCET measured for transmit direction of the network stack. For receive direction, the difference is about 28,000 cycles. For PDU sizes of 1500, this is 50% of the WCET. Figure 9 shows the execution times for correctly flooded cache, the execution times when simply writing into the cache, and the unflooded case. The applications share the cache in all cases.

Especially the receive path makes clear that scheduling should respect the influence of cache accesses to execution times. While a large amount of time can be bound with cache partitioning, the influence of the L1 cache remains. One solution could be to schedule real-time components as non-interruptible parts. The sum of worst-case execution times of these parts is subject to reservation. The problem is that IRQ acceptance would be delayed. When dealing with hardware which handles only single transactions and an interrupt signals the end of each transaction, delaying IRQ acceptance will slow down the system enormously. A better solution for scheduling IRQs is to ascertain the maximum influence of the IRQ to other resources, e.g., to caches. We could assume an IRQ to bring the cache in the worst case state regarding the network stack. The amount of time the network stack execution is extended by unfavorable cache state must be reserved additionally for every IRQ occurrence. Essential to this reservation is the knowledge of the hardware driver about the IRQ behaviour of its hardware.

5 Conclusion

This paper presented algorithms to achieve worst case situations for subsequent memory accesses.

Based on that, worst case execution times of operations of the network stack were determined. It was shown, that cache partitioning significantly improves the worst case execution times of the stack, which results in a more appropriate resource reservation and a better system utilization. The maximum overhead caused by cache misses is reduced from 310% to 90% for receiving PDUs of typical sizes and from 78% to 26% for the transmit direction.

It was also shown, that achieving the worst case requires intense knowledge of the underlying system. Results obtained from wrong assumptions to cache architecture or from wrong cache flooding techniques differ from the real worst case by up to 50%.

Acknowledgements

We want to tank Jochen Liedtke for discussion about the cache hierarchy and pointing out the real worst cases with set-associative L1 caches. We also thank Sebastian Schönberg for proofreading and his valuable comments.

References

[BH98] Martin Borriss and Hermann Härtig. Design and implementation of a real-time ATM-based protocol server. In *19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[BW88] J. L. Baer and W. H. Wang. On the inclusion properties for multi level cache hierarchies. In *15th Annual International Symposium on Computer Architecture (ISCA)*, pages 73–80, Honolulu, HA, June 1988.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[LHH97] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[Lie95] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Mue95] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, June 1995.

[WBL89] W. H. Wang, J. L. Baer, and H. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *16th Annual International Symposium on Computer Architecture (ISCA)*, pages 140–148, Jerusalem, May 1989.

[Wol93] A. Wolfe. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*, September 1993.