When Spatial and Temporal Locality Collide: The Case of the Missing Cache Hits

[Experience Paper]

Mattias De Wael
Software Languages Lab
Vrije Universiteit Brussel
madewael@vub.ac.be

David Ungar
Watson Research Center
IBM Research
dungar@ibm.com

Tom Van Cutsem
Software Languages Lab
Vrije Universiteit Brussel
tvcutsem@vub.ac.be

ABSTRACT

Even the simplest hardware, running the simplest programs, can behave in the strangest of ways. Tracking down the cause of a performance anomaly without the complete hardware reference of a processor is a prime example of black-box architectural exploration. When doubling the work of a simple benchmark program, that was run on a single core of Tilera's TILEPro64 processor, did not double the number of consumed cycles, a mystery was unveiled. After ruling out different levels of optimization for the two programs, a cycle-accurate simulation attributed the sub-optimal performance to an abnormally high number of L1 data cache misses. Further investigation showed that the processor stalled on every Read-After-Write instruction sequence when the following two conditions were met: 1) there are 0 or 1 instructions between the write and the read instruction and 2) the read and the write instructions target distinct memory locations that share an L1 cache line. We call this performance pitfall a *RAW hiccup*. We describe two countermeasures, memory padding and the explicit introduction of pipeline bubbles, that sidestep the RAW hiccup.

This experience paper serves as a useful troubleshooting guide for uncovering anomalous performance issues when the hardware design under study is unavailable.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

Keywords

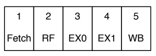
L1 data cache, TilePro64, padding, pipeline bubble

1. INTRODUCTION

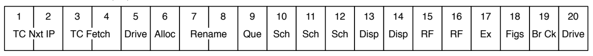
In the context of our research in parallel algorithms for many-core architectures, we chose to explore the TilePro64 processor by writing a simple program and measuring its absolute and relative performance on a single core. To verify the measured results, we doubled the work of our program expecting the running time to double as well.

The experiment, however, revealed that *duplicating the work did not double the execution time.* What could possibly be causing this anomalous performance? And how does one track down such changes in performance efficiency? We conducted a series of experiments, timing and simulating different code sequences. Each step answered one question only to ask another. Finally, we were able to pinpoint the instruction and memory reference sequence that was responsible. Our hunt for the performance anomaly is a nice example of black-box architectural exploration, as we did not have a access to a complete hardware design reference.

The rest of this paper is organized as follows: First the used hardware is discussed, followed by the presentation of the two benchmark programs that are used throughout the text. Section 4 elaborates on the *expected* and *measured* performance of both programs, revealing a significant discrepancy between the two. In Section 5 the compiler generated instruction sequences of the benchmark programs are discussed. Section 6 focusses on processors stalls and how they map to the source code. Section 7 and 8 each describe a countermeasure that sidesteps the RAW hiccup in the benchmark programs. Finally, in Section 9 we compare the RAW hiccup to similar performance pitfalls.

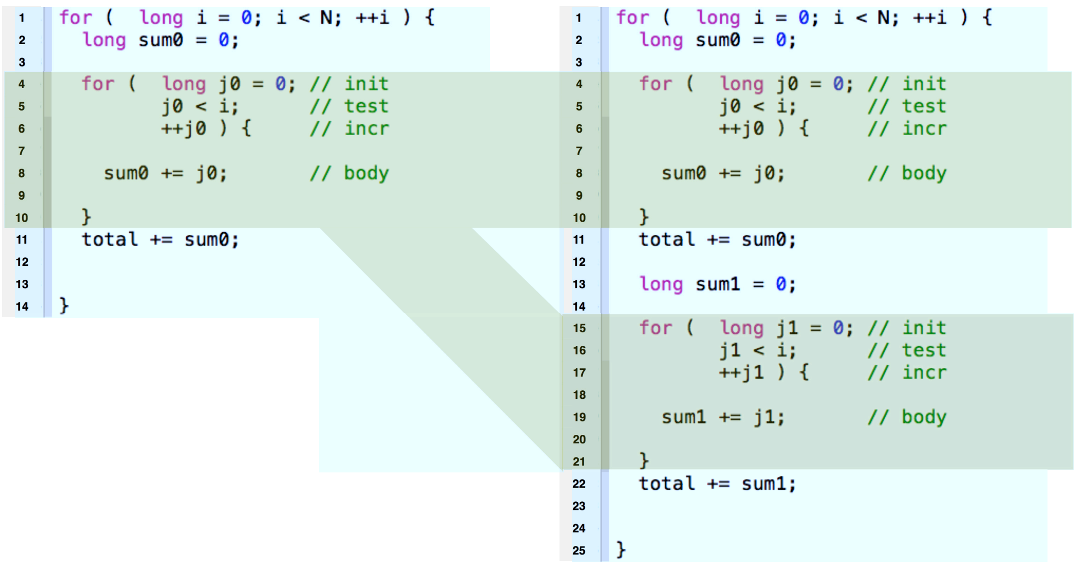


(a) 5 Stage Pipeline of the TILEPro64



(b) 20 Stage pipeline of the Intel Pentium IV.

Figure 1: The Tilera TilePro64 processor has a much more shallow pipeline than for instance Intel's Pentium IV. Therefore, understanding programs and predicting instruction timings on the TilePro64 processor should be relatively easy



```
1  for (  long i = 0; i < N; ++i ) {
2    long sum0 = 0;
3
4    for (  long j0 = 0; // init
5           j0 < i;      // test
6           ++j0 ) {     // incr
7
8      sum0 += j0;       // body
9
10   }
11   total += sum0;
12
13
14 }
```

```
1  for (  long i = 0; i < N; ++i ) {
2    long sum0 = 0;
3
4    for (  long j0 = 0; // init
5           j0 < i;      // test
6           ++j0 ) {     // incr
7
8      sum0 += j0;       // body
9
10   }
11   total += sum0;
12
13   long sum1 = 0;
14
15   for (  long j1 = 0; // init
16          j1 < i;      // test
17          ++j1 ) {     // incr
18
19     sum1 += j1;       // body
20
21   }
22   total += sum1;
23
24
25 }
```

Figure 2: Two simple benchmark programs, Program 1 and Program 2 respectively, where the second program has exactly twice the work to do compared to the first program. Program 2 is expected to consume twice as many cycles as Program 1.

2. THE PLATFORM: TILEPRO64 PROCESSOR

At first sight, Tilera's TILEPro64 [9] might look like a complex many-core processor chip. Its 8x8 mesh network connects 64 processing cores, keeps two levels of distributed cache coherent, and supports inter-core communication. Each core utilizes a three-way Very Long Instruction Word (VLIW) architecture to support explicit Instruction Level Parallelism (ILP) by executing up to three *bundled instructions* simultaneously by one of the three pipelines of a single core.

But, when looked at in isolation, a single pipeline of a single core of the TILEPro64 processor has a relatively simple architecture. The in-order pipelines execute an instruction in 5 stages (Figure 1a): a rather shallow pipeline compared to contemporary hardware that have pipelines as deep as 20 stages (Figure 1b) [5]. With this short, in-order pipeline the TILEPro64 aims at low branch and low load-to-use latencies. Thus when considering a single pipeline on a single core, the TILEPro64 can arguably be described as simple hardware and performance prediction of any sequential programs it runs should be fairly easy. But even the simplest hardware running the simplest programs, can behave in the strangest of ways. How simple could those programs be?

3. THE PROGRAM: TETRAHEDRAL NUMBERS

We chose a very simple algorithm for our benchmark: the computation of the nth tetrahedral number. To compute the nth tetrahedral number it suffices to accumulate the n first triangular numbers [7]. And to compute the nth triangular number it suffices to accumulate the n first non-zero integers [8]. Mathematically this could be written as $\sum_{i=0}^{N} \sum_{j=0}^{i} j$ and Program 1 (Figure 2) is the straightforward translation of this formula into C-code. Thus, Program 1 consists of one outer loop (lines 1-14 in Figure 2) with one inner loop (lines 4-10 in Figure 2), of which the body consists of a single statement accumulating a counter (line 8 in Figure 2). Program 2 (Figure 2) just about doubles the work of Program 1 by repeating the inner loop (lines 4-10 and lines 15-21 in Figure 2). When measurements revealed that *doubling the work did not double the execution time*, we were mystified.

4. MEASURED PERFORMANCE

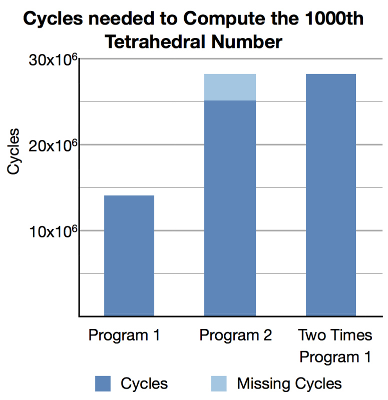


Figure 3: Cycles consumed by Program 1 and Program 2, compared to the number of expected cycles for Program 2.

"To measure is to know", but in this case measuring the performance of both benchmark programs raised more questions than it resolved. We measured the absolute performance of both programs in *cycles needed to complete the outer loop*. Computing the 1000th tetrahedral number by running Program 1 requires 14×10^6 cycles. Doubling the work by running Program 1 twice, requires 28×10^6 cycles. Surprisingly, when running Program 2 which also does twice the work as Program 1, only 25×10^6 cycles were consumed. The difference of 3×10^6 cycles between expected and measured performance, depicted as the lighter colored box in 3,

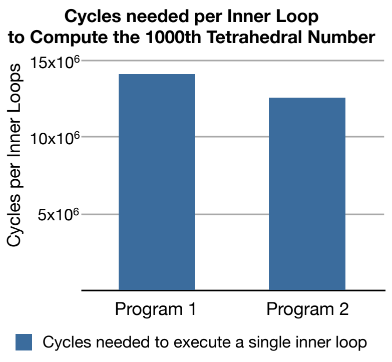


Figure 4: Average number of cycles needed by Program 1 and Program 2 to compute a single inner loop. Surprisingly, Program 2 needs fewer cycles than Program 1.

is too large to be attributed to the overhead of running two outer loops when running Program 1 twice.

Put differently, as depicted in Figure 4, when looking at the average number of cycles needed to complete a single inner loop, we see that Program 1 needs 14×10^6 cycles, as opposed to the 12.5×10^6 cycles needed by Program 2. These data suggest that the loops in Program 2 run more efficiently. But why would the same source code run faster?

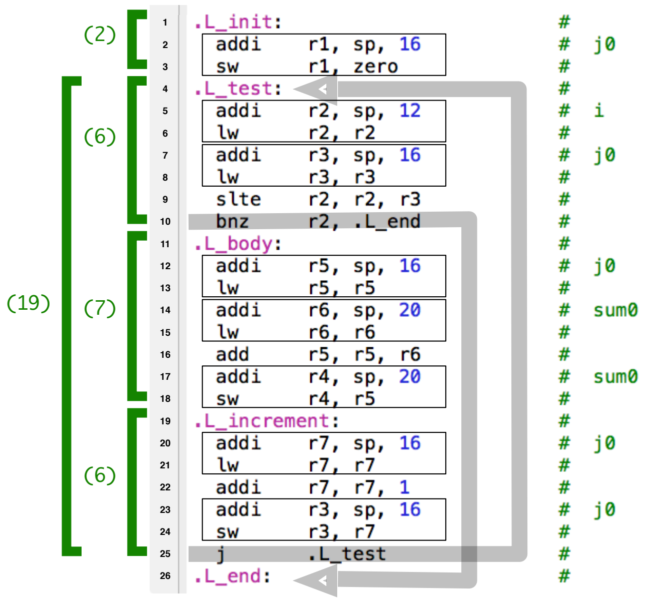
5. COMPARING INSTRUCTION SEQUENCES

The execution of the inner loops require quadratic time, actually the number of additions needed to compute the nth tetrahedral number is equal to the nth triangular number, and thus indeed *quadratic* in function of N. Since the outer loops induce only linear overhead, *the inner loops dominate the overall performance* of the benchmark programs, and we focused on these.

Could it be that the compiler was generating different instruction sequences for syntactically equal inner loops? If this were true, the assumption that Program 2 did exactly twice the work of Program 1 would be false, and the unexpected execution times could be explained. So the first step in unraveling the mystery was to guarantee that the generated machine instruction sequences for the two programs were similar for all inner loops.

We used Tilera's gcc compiler 2.0.2 with optimization flag -O0 which compiled the inner loop of Program 1 into the 21 instructions shown in Figure 5. The first two instructions initialize the inner loop and are only executed a linear number of times. More interesting were the remaining 19 instructions, which formed the heart of the computation and were executed $\frac{N\times(N-1)}{2}$ times each. Ignoring the small constant and linear overhead induced by the inner and outer loops, $19 \times \frac{N\times(N-1)}{2}$ was a fair approximation of the total number of executed instructions for Program 1 when N was sufficiently large.

This instruction sequence can be further optimized. The most invasive and effective optimization would be to reduce the program to compute the formula $\frac{N\times(N+1)\times(N+2)}{N}$ which would be semantically equivalent to Program 1 but would require only constant time for any input. But even if we wanted to keep the structure of the computation, many optimizations were possible. For example, the computation



```
.L_init:                  #
    addi   r1, sp, 16     #  j0
    sw     r1, zero       #
.L_test:                  #
    addi   r2, sp, 12     #  i
    lw     r2, r2         #
    addi   r3, sp, 16     #  j0
    lw     r3, r3         #
    slte   r2, r2, r3     #
    bnz    r2, .L_end     #
.L_body:                  #
    addi   r5, sp, 16     #  j0
    lw     r5, r5         #
    addi   r6, sp, 20     #  sum0
    lw     r6, r6         #
    add    r5, r5, r6     #
    addi   r4, sp, 20     #  sum0
    sw     r4, r5         #
.L_increment:             #
    addi   r7, sp, 16     #  j0
    lw     r7, r7         #
    addi   r7, r7, 1      #
    addi   r3, sp, 16     #  j0
    sw     r3, r7         #
    j      .L_test        #
.L_end:                   #
```

Figure 5: The inner loop of Program 1 compiled into assembler code (-O0). The last 19 instructions dominate the performance of Program 1, and are executed $\frac{N\times(N-1)}{2}$ times.

in Program 1 used only 4 variables and the value of the stack pointer (register) is not changed during the execution. Therefore it would be possible to compute the absolute addresses of the four variables once and store them in a dedicated register. This optimization could save 7 addi instructions in each iteration. The difference in relative performance would be accounted for if the compiler had applied this or any other optimization to the inner loops of Program 2, but not to the inner loop of Program 1. But a comparison of the generated assembler instruction sequences for both Program 1 and Program 2 revealed that *all three inner loops were compiled into the same 21 instructions.* The instruction sequences for each loop differed only in the relative addresses of the variables. At this point, it was clear that twice as many assembler instructions were executed for Program 2, than for Program 1. Yet, Program 2 was executing those instructions more efficiently. What could explain this behavior?

If an instruction sequence gets more densely packed into instruction bundles such that the instruction level parallelism supported by the VLIW architecture gets exploited, it will run faster. Could it be possible that the instructions of Program 2 got bundled more efficiently than those for Program 1? The assembly-level instruction sequence, as shown in Figure 5, did not make these bundles explicit. And since the processor cores operate in-order they are not responsible for any implicit instruction level parallelism themselves. When we decompiled the machine instructions back into assembler code we saw that all instructions were wrapped in a single bundle and thus issued to be executed sequentially without any instruction level parallelism. *Bundles were not the answer.*

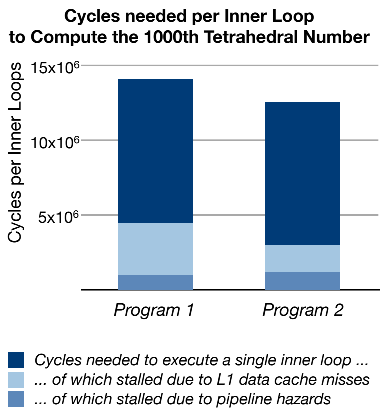


Figure 6: Relative to the amount of work, Program 1 suffered from many more L1 data cache misses than Program 2. This discrepancy explained the difference in performance, but left us wondering about the cause of the discrepancy.

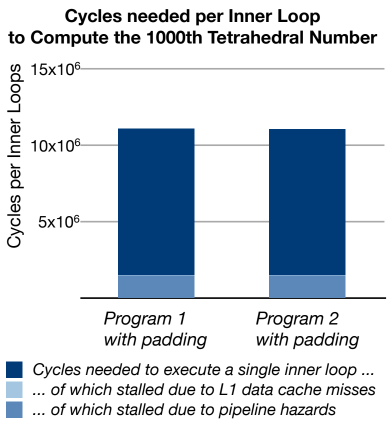


Figure 8: Adaptations of Program 1 and Program 2 that add padding around the inner loop counters make all L1 data cache misses disappear. Consequently, also the change in efficiency is gone and both programs behave as expected.

Because all executed bundles were single-instructions, the processor executed the exact same instruction sequence once for Program 1 and twice for Program 2. Why did the processor work at different speeds in each program?

6. CACHE MISSES CAUSED THE PROCESSOR TO STALL

If a processor consumes a different amount of cycles for equal instruction sequences, it must be stalling somewhere. A cycle-accurate simulated execution of the applications reported that the processor was indeed stalling, and attributed these stalls to L1 and L2 instruction- and data-cache misses, to pipeline hazards, and to mispredicted branches. Of these categories, only the *L1 data cache misses* and the *pipeline hazards* were numerous enough to observably impact the performance. If these stalls were the true culprits, then the anomalous increase in efficiency of Program 2 over Program 1 should also have been reflected in the number of corresponding stalls. For stalls attributed to pipeline hazards, doubling the work also doubled the number of stalls. A regular evolution, so that category was exonerated. Remarkably, *the number of stalls caused by L1 data cache misses was not affected by doubling the work.*

Even considering each program in isolation, the number of L1 data cache misses was unexpectedly high. Because both benchmark programs used only 4 and 6 variables respectively, we *expected no misses* at all. But the clue to the mystery was in the observation that the number of L1 data cache misses did not increase when the amount of work was doubled. Figure 6 shows these data relative to the number of inner loops: on average a single inner loop of Program 2 suffered from half as many L1 data cache stalls as the loop of Program 1. Why was Program 2 more efficient?

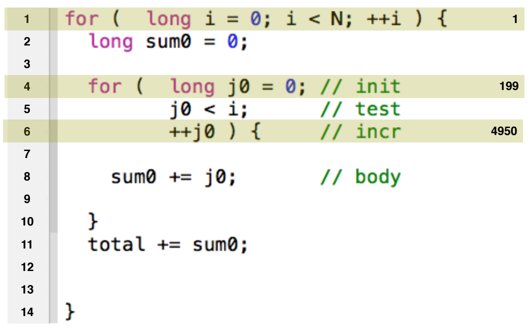
More rigorous simulation revealed that all the L1 data cache misses were read misses. This commonality limited the instructions where the misses could occur to the lw (load word) instructions of the program (lines 6, 8, 13, 15, and 21 in Figure 5). On the C-code level, the cycle-accurate simulator indicated that the loop-increment operations on $j0$ (line 6 in Figure 7a) in Program 1, and on $j1$ (line 17 in Figure 7b) in Program 2 were *responsible for almost all stalls*, as is shown in Figure 7 where the source lines of Program 1 and Program 2 are annotated with the number of observed cache misses. The synthesis of these two pieces of evidence allowed us to identify the slow instruction in each program. In Program 1 this would be the load instruction lw r7 r7 shown on line 21 of Figure 5. When N is 1000, the number of inner loop iterations is 4950 which was exactly the number of observed L1 data cache misses on line 6 and 17 of Program 1 and Program 2 respectively.

What we know now is that the inner loop of Program 1 ran slowly because *every iteration induced a cache miss.* One of the inner loops of Program 2 ran equally slowly for the same reason, but on average Program 2 was more efficient because the other inner loop never suffered from a cache miss. But why did the processor have to wait for data that should have been in its L1 cache in the first place?

7. PADDING RESOLVES THE CACHE MISSES

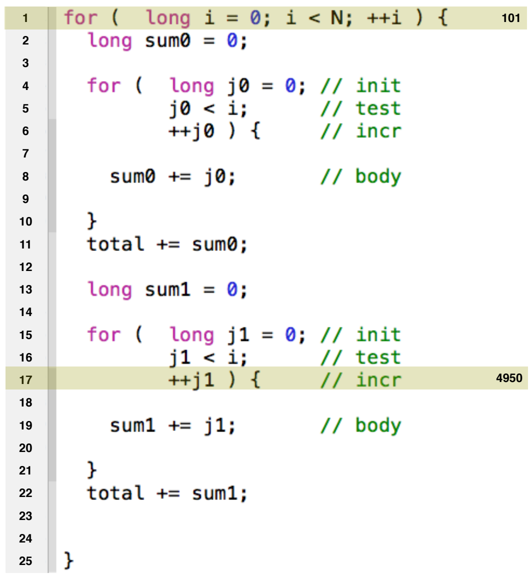
Recall that the difference in performance of our two benchmark programs was caused by an abnormally large number of L1 data cache misses that were not expected and moreover did not increase with the amount of work. A cache miss occurs when a processing unit fails to access a piece of data in the cache which results in a much more expensive operation that reroutes the memory instruction to the next level of memory. In this case reading from the L1 data cache fails, causing a load from the L2 cache which is 7 cycles away on the TILEPro64.

Consider Program 1, which used only 4 variables for its entire computation. By the time the instruction on line 15 in Figure 5 had been executed for the first time, we expected all the variables to reside in the L1 cache and to stay there for the remainder of the computation. This expectation arose because 8KB of L1 data cache is plenty of room for storing 4 values of 4 bytes each: they even fit on a single cache line as the TILEPro64's L1 cache lines are 16 bytes wide [9]. Even for the 6 variables of Program 2 the 8KB should have more than sufficed. Thus any L1 data cache line that was



```
for (  long i = 0; i < N; ++i ) {                 1
  long sum0 = 0;

  for (  long j0 = 0; // init                   199
         j0 < i;      // test
         ++j0 ) {     // incr                   4950

    sum0 += j0;       // body

  }
  total += sum0;


}
```

(a) Program 1: The bulk of L1 data cache misses occur on the increment of the only inner loop.



```
for (  long i = 0; i < N; ++i ) {               101
  long sum0 = 0;

  for (  long j0 = 0; // init
         j0 < i;      // test
         ++j0 ) {     // incr

    sum0 += j0;       // body

  }
  total += sum0;

  long sum1 = 0;

  for (  long j1 = 0; // init
         j1 < i;      // test
         ++j1 ) {     // incr                   4950

    sum1 += j1;       // body

  }
  total += sum1;


}
```

(b) Program 2: The bulk of L1 data cache misses occur on the increment of the second inner loop.

Figure 7: To understand the reason for all these cache misses, a better view is needed on where they actually occur. Program 2 has one normal and one affected loop. Since these are syntactically equivalent, memory layout of variables is a probable culprit of the discrepancy.

invalidated in the execution of either program qualified as unanticipated behavior.

In parallel computing, a case of unanticipated cache invalidation, called *false sharing* occurs when a computation writes to a memory location that resides on the same cache line of a distinct memory location used by a concurrent computation [6]. False sharing can be reduced by padding the memory layout of variables so that the memory locations used by concurrent computations do not share cache lines [1]. Padding is a low-level programming technique in which, usually unused, memory is allocated around variables to obtain a more suitable layout of variables in the caches and/or memory. Although this line of inquiry seems far-fetched in the case of a sequential program using only 4 variables, we experimented with memory layout anyway because the distributions of variables over cache lines can affect the number of misses.

Figure 9 shows four possible layouts of the variables over different cache lines when padding is introduced. Each line shows three cache lines (alternating colors) of four times 4 bytes each. The first line in Figure 9 shows the memory layout as observed for Program 1 without any padding: the first cache line (white) contains the variables *total* and i, the consecutive cache line (gray) contains the variables $sum0$ and $j0$, finally the third cache line (white) contains no variables relevant to our case. The others three layouts in Figure 9 show how introducing padding before $j0$ moved $j0$ into a different cache line than the other 3 variables.

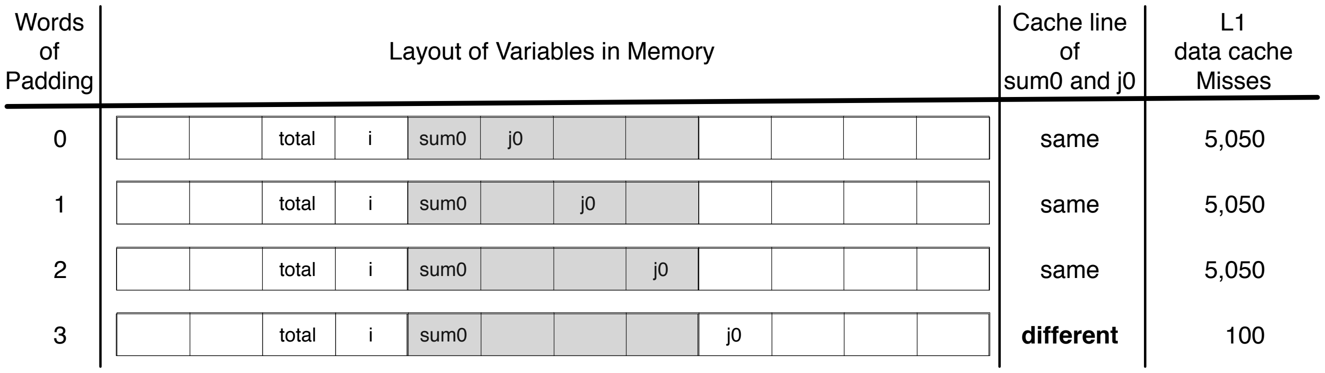
Program 1, when adapted such that $j0$ resided in its own cache line, consumed only 11×10^6 cycles, as opposed to the 14×10^6 cycles it had consumed before. The cycle-accurate simulation showed that the *padding eliminated all the L1 data cache misses.* Further, when we moved the variable $j1$ of Program 2 to a separate cache line, the number of cycles consumed by Program 2 dropped from 25×10^6 to 22×10^6 . The import of this performance improvement is that neither benchmark program suffered from unanticipated L1 data cache misses when a change in the memory layout places the inner loop counter and the inner loop accumulator on different L1 data cache lines.

Summarized, the lw instruction (line 21 in Figure 5) causes the processor to stall if the preceding sw instruction (line 18 in Figure 5) targets the same L1 data cache line. For this reason we call this behavior a *read-after-write hiccup*, or RAW hiccup for short. In our example, the sw and lw instructions are separated by an addi instruction (line 20 in Figure 5) which raised the question if, besides the memory layout, also the addi instruction plays a role in the RAW hiccup?

8. INJECTING PIPELINE BUBBLES

Isolating the missed variable in a different cache line sidestepped the RAW hiccup, but could instruction reordering accomplish the same?

Data hazards occur when subsequent instructions have data dependencies and are executed at the same time in a pipeline. Many contemporary processors use out-of-order execution to avoid these dependencies [4]. The TILEPro64 processor, however, supports only in-order execution. In that case the only way to avoid the data hazard is by introducing a pipeline bubble. Pipeline bubbling is an instruction scheduling technique that prevents data hazards from occurring by delaying the execution of dependent instructions in the pipeline. Typically this is done by the processor's logic by stalling the execution of the depending instruction. A compiler could simulate this behavior by inserting a no-operation instruction (NOP), but what if we hardcoded such a NOP in the slow running inner loops of Program 1 and Program 2?



Words of Padding	Layout of Variables in Memory												Cache line of sum0 and j0	L1 data cache Misses
0			total	i	sum0	j0							same	5,050
1			total	i	sum0		j0						same	5,050
2			total	i	sum0			j0					same	5,050
3			total	i	sum0				j0				**different**	100

Figure 9: With as few as 4 variables there is no reason to suffer from cache eviction. In parallel computing padding is a tried and true approach to tackle false sharing, a phenomenon where cache eviction is also unanticipated. Adding padding between $sum0$ and $j0$, such that they reside on different cache lines, causes a significant drop in L1 data cache misses.

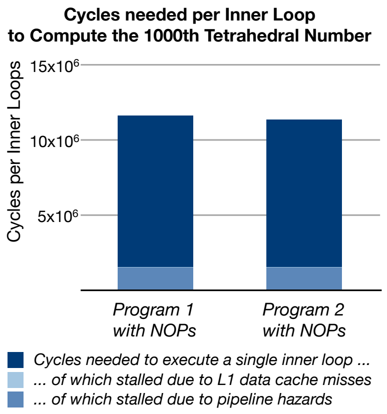
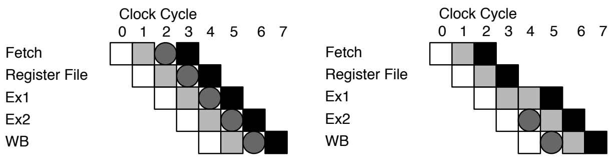


Figure 10: Adding NOPs to the bodies of the slow inner loops of Program 1 and Program 2 make all L1 data cache misses disappear. Consequently, also the change in efficiency is gone and both programs behave as expected.



(a) Pipelined execution of 3 instructions where the processor stalls on the Ex1 stage of the second instruction.

(b) Pipelined execution of 2 instructions, a NOP, and another instruction where the processor does not stall.

Figure 11: The overall performance of a processor stalling for one cycle in a sequence of three instructions, is equivalent to the performance of executing that same sequence with an additional NOP instruction if the processor does not stall.

In our benchmark programs there was no actual data dependency between the memory instructions on lines 18 and 21 (see Figure 5), the targeted memory locations only shared a cache line. Thus, adding a NOP in the body of the slow running inner loop should have made the program run even slower because the inner loop now consisted out of 20 instead of 19 instructions. Perversely, the measured performance was closer to that of the fast programs with padding, than to the execution time of the original benchmark programs with all the cache misses. The cycle-accurate simulator indicated that *adding a NOP to the body of the slow inner loops removed all L1 data cache misses.*

Synthesizing the effects observed when either padding or pipeline bubbles were introduced, allowed us to conclude that *on the TILEPro64 a RAW hiccup occurs in a read-after-write instruction sequence if two conditions are met. First, if both the store and the load instruction target distinct memory locations that share an L1 data cache line, and second if at most one instruction separates the store and the load.* If either condition is eliminated, the RAW hiccup is gone. To overcome this pitfall on the hardware level, a much more complicated power and area consuming micro-architecture would be needed in order to work around the store word inefficiencies.

9. SIMILAR PERFORMANCE PITFALLS

From the software perspective, pinpointing the origin of anomalous performance to a specific instruction and memory reference sequence suffices to render a program more performant simply by avoiding that sequence. From a hardware perspective, the question remains what architectural design choices caused the anomalous performance. To the best of our knowledge the details of hardware implementations we are hitting in this concrete example are not documented by the chip producer. Without these details we can only make an educated guess about the concrete origin of the stalls.

However, three well know performance pitfalls exist that look similar to the RAW hiccup:*false sharing*, *load-hit-store*, and *write misses.*

False sharing *False sharing* only occurs in the case of concurrent processes, and thus does not apply in the case of a RAW hiccup which occurs in a single thread of control [1]. But, besides the number of processes the commonalities are omnipresent: both performance pitfalls occur when a sequence of memory instructions, with at least one write, target unrelated variables that share a cache line.

Load-hit-Store A single process can suffer from the performance pitfall *load-hit-store* (LHS) when a read is issued too soon after a write to read the new value [3]. The RAW hiccup also issues a read too soon after a write, but as opposed to a LHS, the memory instructions target *distinct memory locations*.

Write Stall Finally, when a processor must wait for writes to complete during *write through*, the processor is said to *write stall* [4]. The L1 data cache of the TILEPro64 uses the *write-through policy*, thus what we called a RAW hiccup could actually be a write stall. Other literature, however, refines the definition of write stalls in the context of write-through caches as the delay caused when *a write encounters another write in progress* [2]. Thus excluding read instructions as the origin of a write stall. This makes the instruction sequence causing a RAW hiccup different from the instruction sequence causing a write stall.

Identifying the actual implementation details that caused the RAW hiccup is notoriously difficult without a complete hardware reference. We recognized *false sharing*, *load-hit-stores*, and *write misses* as performance pitfalls similar to the RAW hiccup. They differ, however, from the RAW hiccup in either number of processes, targeted memory locations, or instruction sequence.

10. CONCLUSIONS

A simple C program exhibited a significant discrepancy between its expected and measured performance. The origin of the anomaly was found to be a counter-intuitive multi-cycle processor stall that occurred whenever a store instruction was followed within two instructions by a load instruction targeting the same L1 data cache line. We called this performance pitfall a RAW hiccup. Eliminating either of the two conditions caused the RAW hiccup to disappear: The introduction of sufficient padding on the one hand and the introduction of a manual pipeline bubble on the other hand removed all anomalous L1 data cache misses.

This experience paper reports on the hunt for a performance anomaly observed when studying the behavior of a trivially simple sequential program. The exact instruction and memory reference sequence that was responsible was found only after performing various experiments and simulations each one a step down the ladder of abstraction. This experience raises the question that if we can not predict the performance of trivially simple sequential code on a simple processing unit, how can we hope to understand the performance of parallel applications running on an 8x8 mesh network-on-chip?

11. ACKNOWLEDGMENTS

Mattias De Wael is supported by a doctoral scholarship granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen), Belgium.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO).

This research has been partially supported by IBM Research.

12. REFERENCES

[1] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *ICPP (1)*, pages 377–381, 1991.

[2] J. S. Emer and D. W. Clark. A characterization of processor performance in the vax-11/780. *SIGARCH Comput. Archit. News*, 12(3):301–310, Jan. 1984.

[3] B. Heineman. Common performance issues in game programming, June 2008.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[5] G. Hinton, D. Sager, M. Upton, D. Boggs, D. P. Group, and I. Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[6] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[7] N. J. A. Sloane. Tetrahedral (or triangular pyramidal) numbers, Oct. 2012.

[8] N. J. A. Sloane. Triangular numbers, Oct. 2012.

[9] Tilera. *Tile Processor User Architecture Manual*, June 2010.

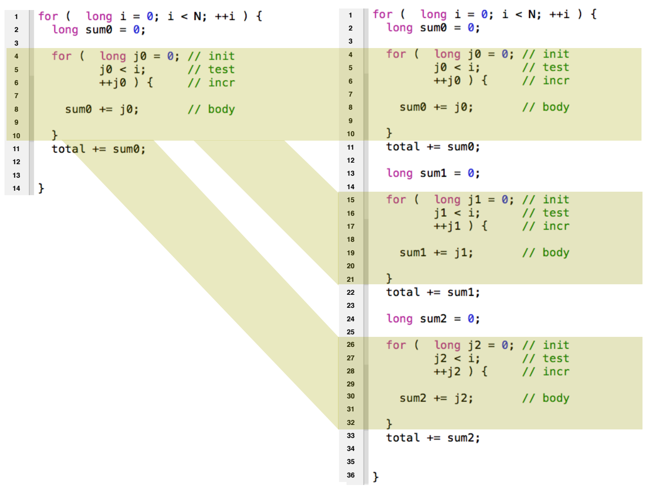
APPENDIX

A. TRIPLING THE WORK

Besides the experiments with Program 1 and Program 2, those presented in this paper, six similar programs also contributed to the discovery of the RAW hiccup. Program 2 was our attempt to double the work of Program 1 by duplicating its single inner loop, the remaining four programs were our attempt to multiply the work of Program 1 by 3 trough 8, also by replicating the inner loop. Figure 12 shows what Program 3 looks like.

Recall that the first clue towards the discovery of the RAW hiccup was that number of L1 data cache misses did not increase when the amount of work was doubled. However, when we *tripled the work* of Program 1 *the number of L1 data cache misses did double*. Adding a fourth inner loop again left the number of L1 data cache misses unchanged. This step-wise increase continued for all eight programs, as is shown in Figure 13. The number of cycles stalled because of pipeline hazards on the other hand increased linearly with the number of inner loops. When considering these data relative to the number of inner loops, as is shown in Figure 14, we see that consecutive programs alternate between being less efficient and more efficient.

At the level of C-source code, this translates to one inner loop suffering from the RAW hiccup for every two inner loops in the program ($\lceil \frac{\#\text{inner loops}}{2} \rceil$). Since all inner loops are compiled into the same instruction sequences (Figure 5) the memory layout must be causing the RAW hiccup. Consider a layout of variables where $sum0$ and $j0$ share a cache line as in line 2 of Figure 9. Introducing a new inner loop consequently introduces also two new variables (e.g. $sum1$ and $j1$) that do not share a cache line. Again adding two new variables (e.g. $sum2$ and $j2$) gives rise to a new RAW hiccup because $sum2$ and $j2$ do share a L1 data cache line. This explains why the number of cache misses only increases when adding a new inner loop to a program with an even number of inner loops.



```
for (  long i = 0; i < N; ++i ) {
  long sum0 = 0;

  for (  long j0 = 0; // init
         j0 < i;      // test
         ++j0 ) {     // incr

    sum0 += j0;       // body

  }
  total += sum0;


}
```

```
for (  long i = 0; i < N; ++i ) {
  long sum0 = 0;

  for (  long j0 = 0; // init
         j0 < i;      // test
         ++j0 ) {     // incr

    sum0 += j0;       // body

  }
  total += sum0;

  long sum1 = 0;

  for (  long j1 = 0; // init
         j1 < i;      // test
         ++j1 ) {     // incr

    sum1 += j1;       // body

  }
  total += sum1;

  long sum2 = 0;

  for (  long j2 = 0; // init
         j2 < i;      // test
         ++j2 ) {     // incr

    sum2 += j2;       // body

  }
  total += sum2;


}
```

Figure 12: Program 3 (right) triples the work of Program 1 (left).

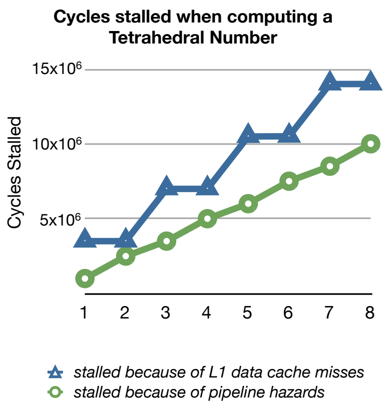


Figure 13: The number of cycles stalled caused by pipeline hazards increases linearly with the amount of work. Remarkably, the number of stalls caused by L1 data cache misses increases in steps.

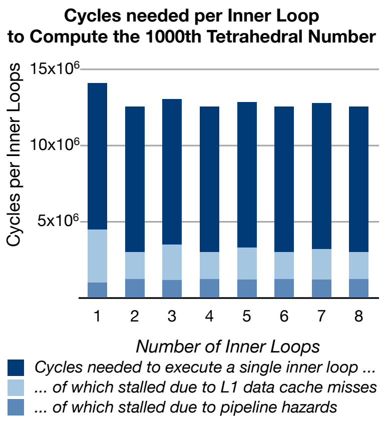


Figure 14: Comparing number of cycles needed to compute a single inner loop shows that programs with an even number of inner loops are more efficient because they suffer from less L1 data cache misses..