Applying the Roofline Model

Master Thesis

by

Ruedi Steinmann

Departement of Computer Science

ETH Zürich

Advisers:	Prof. Markus Püschel
	Georg Ofenbeck
Institute:	Institute for Computer Systems
Group:	Advanced Computing Laboratory

April 30, 2012

Abstract

In the beginning, CPU performance was mainly improved by increasing the core frequency. This approach is limited by several physical issues. These limits were reached about one decade ago. Since then performance improvements have been achieved at the cost of a strong increase in CPU complexity. This raises the difficulty to understand their behavior, which is important for software performance optimization.

In this context, insightful yet easy to understand performance models are of great value. The Roofline Model premises to fulfill these criteria.

Unfortunately, to our best knowledge there is no tool available to measure the required quantities and generate Roofline Plots. We built such a tool, evaluated the quality of it's measurement results and generated Roofline Plots of various code samples.

The tool is freely available and intended to be used by a wider audience.

Contents

Acknowledgements

First I would like thank Georg Ofenbeck for supervising my thesis. He guided me through the process of writing this thesis. In addition he proof-read my drafts and provided very valuable feedback.

I worked in the group of Professor Markus Püschel. He creates a very positive working atmosphere among his staff. We held few but very productive meetings. They always helped me to see clear again.

I am thankful to Daniele Spampinato and Zoltán Majó for all the interesting discussions we had.

Finally, I would like to thank my wife. She was always there cheering me up and never letting me forget to work on the documentation.

Chapter 1

Introduction

‘The free lunch is over’ [15] In the past decades, the CPU manufactures were reliably able to increase the clock speed. However, this has become harder and harder due to not just one but several physical issues. To increase performance despite the limited clock speed, the CPUs become more and more complex.

From the perspective of software performance optimization, the increased complexity made performance observations harder to understand. Since the trend to more complex CPUs is likely to continue in the future, an insightful and easy-to-understand performance model would be of high value.

The roofline model [21] promises to fulfill these criteria. The key observation which led to this model is that, for the foreseeable future, the transfer bandwidth between CPU and memory will often be the limiting factor of system performance. This is called the memory bottleneck.

In order to use the arithmetic units found in the CPU close to their peak performance, some of the data loaded from memory has to be used in multiple operations. This is enabled by the caches found in all modern computer systems. A cache is a relatively small memory area which has a low access latency and a high bandwidth. All data loaded from memory is saved in the cache, evicting the least recently used data present in the cache.

If the implementation of an algorithm manages to largely operate on data which can be held in the CPU caches, off-chip bandwidth will not be an issue. Unfortunately it is often difficult to organize the data access pattern of an algorithm in such a way that repeatedly accessed data is not wiped out of the cache by intermediate data accesses.

During the optimization of an algorithm, many changes have an effect on both performance and data access pattern. It is therefore beneficial for a performance model to capture both aspects.

Towards this goal, the roofline model introduces the term ‘operational intensity’ defined as the number of performed operations per byte transferred between CPU and memory. The exact definition of operation in the context of this definition is problem specific, but floating point operation is often used.

The roofline model combines the operational intensity on the x axis with the performance on the y axis in a 2D graph . Each measurement results in a point on the plot. If a measurement is repeated for different problem sizes, the resulting points in the roofline plot can be connected by a line to a series.

The peak floating point performance appears as a horizontal line. There can be different bounds depending on the technology or the number of cores used.

The memory bandwidth can be shown in the roofline plot, too. If the implementation of an algorithm transfers v bytes (transfer volume) and performs Ω operations within t cycles (the unit of time does not matter, could be seconds as well), it will be placed at $x = \Omega/v$ (operational intensity) and $y = \Omega/t$ (performance). The data throughput between CPU and memory is $\tau = v/t$, which **can be derived from performance and operational intensity**:

$$\tau = \frac{v}{t} = \frac{\Omega v}{\Omega t} = \frac{\Omega}{t}\frac{v}{\Omega} = \frac{\Omega}{t}(\frac{\Omega}{v})^{-1} = yx^{-1} = \frac{y}{x}$$

The throughput is bound by the bandwidth ($\tau < \beta$), which leads to the following limit for performance and operational intensity:

$$\tau = \frac{y}{x} < \beta$$
$$y < x\beta$$

Roofline plots use a log-log scale. This enables the presentation of data in a huge range. In addition, the bandwidth line appears as a line of unit slope, offset towards the top left of the graph proportional to the logarithm of the memory bandwidth. [1]

As for the peak performance, there can be multiple bounds, depending for example on the number of cores used, the memory type or the access pattern (random vs. linear).

The horizontal and diagonal lines, called performance and bandwidth ceilings, give the roofline model it's name. The point where the lines intersect is called the ridge point.

The power of the model comes from the ability to show performance and effects of the data access pattern at the same time. To illustrate this point, we will try to solve the following problem: Given a matrix M of size $n \times m$ (m rows, n columns), we would like to perform the following transformation: $M_{ij} = 3j^2 M_{ij}$.

The straight forward approach is to use the following loop: (Variant 0)

```
for (int i = 0; i < m; i++)
for (int j = 0; j < n; j++)
M[i*n + j] = 3*j*j*M[i*n + j];
```

In an attempt to optimize the algorithm, we could switch the order of the loops to avoid repeated computations of the term $3j^2$: (Variant 1)

[1] In order to grasp the appearance of the bandwidth line, we have to map the line to a cartesian coordinate system . The coordinates x' and y' in the cartesian coordinate system are in the following relationship to the log-log coordinates:

$$x' = \ln(x);\ e^{x'} = x$$
$$y' = \ln(y);\ e^{y'} = y$$

The line becomes:

$$\begin{aligned} y &= \beta x \\ e^{y'} &= \beta e^{x'} \\ y' &= \ln(\beta e^{x'}) \\ &= \ln(\beta) + \ln(e^{x'}) \\ y' &= \ln(\beta) + x' \end{aligned}$$

```
for (int j = 0; j < n; j++) {
   int tmp = 3*j*j;
   for (int i = 0; i < m; i++)
     M[i*n + j] = tmp*M[i*n + j];
}
```

We measured the two variants with $m = 1e6$ and $n = 4$ on an arbitrary computer and got the following performances:

Variant	Performance
0	0.193
1	0.026

Using just this information it is hard to say why our modification degraded performance.

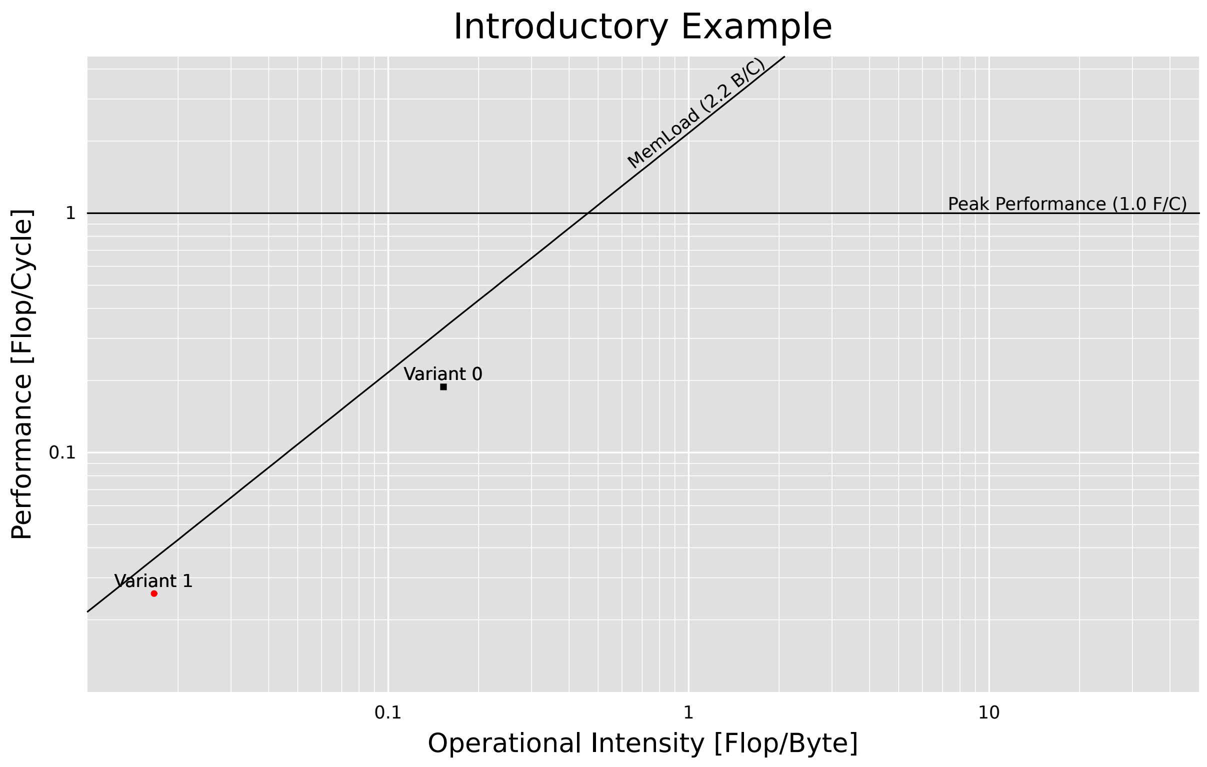


Figure 1.1: Roofline Plot of the Introduction Example

Using the roofline model we get figure 1.1. We see that the operational intensity decreased by an order of magnitude for variant 1. Since both variants operate close to the memory bandwidth, it is clear that the performance has to fall together with the operational intensity. In addition the plot makes clear that the only way to get a substantial speed improvement is to increase the operational intensity. Reviewing our modification, we can see that we moved from a row major to a column major access pattern which is well known to potentially reduce operational intensity.

Generally, if the roofline point of an algorithm lies to the left of the ridge point, the only way to reach peak performance is to increase the operational intensity. On the other hand, if a point lies to the right of the ridge point, memory bandwidth is not the limiting factor. If the kernel does not reach the peak performance, there must be other performance limiting factors.

Apart from guiding the optimization process, the roofline model can be used to analyze different computer systems. When comparing the roofline graphs of different machines, the position of the ridge point is an important clue of how difficult it is to reach a good performance. The further the ridge point lies to the right, the higher is the operational intensity required to achieve peak performance.

1.1 Goals of the Thesis

This thesis aims to create a tool capable of easily creating roofline plots. In addition it should provide a measuring infrastructure which can be used in a broader context.

Code to be measured can come either in the form of a relatively isolated routine, typically containing a single kernel, or in the form of a larger program, containing multiple different kernels. While routines are treated as atomic units, it is desirable to split the larger program into it's different algorithms and do the analysis for each algorithm separately. In the rest of this thesis, we will use the term kernel for a routine or part of a program which is analyzed on it's own.

Many kernels are multi threaded in order to take advantage of CPUs with multiple cores. This is achieved by splitting the algorithm into multiple threads which can be executed in parallel. Our tool should be able to handle such kernels.

To get the data points for a kernel, we need to measure it's performance and the operational intensity.

Performance is defined as amount of work per unit of time. The time required to do the work has to be measured. The amount of work and how it is defined is determined by the actual kernel.

Operational intensity is defined as operations per byte of data traffic. Since the operational intensity cannot be measured directly, we have to measure the operation count and the data traffic.

The point where data traffic is measured is typically between the last level cache of the processor and the DRAM, but it could be measured as well between the different cache levels or between L1 cache and processor.

To obtain the peak performance and memory bandwidth lines for the roofline model we need to create micro benchmarks. They are typically designed to transfer as much data per unit of time for memory bandwidth lines and to execute as may operations per unit of time for peak performance lines as possible. Thus the measurement capabilities needed to evaluate the actual problems can be reused for the benchmarks.

In summary, the following quantities need to be measured:

- Execution Time
- Memory Traffic
- Operation Count

The accuracy and precision of the measurement results will be evaluated and possibly improved by statistical techniques.

The results will be used to automatically generate roofline plots. The results of different kernels and sizes can be combined into a single plot.

The tool is written for the x86 platform and should support different processors.

1.2 Related Work

There are various performance models other than the roofline model. [6] [7] In particular, the 3Cs (compulsory, capacity and conflict misses) model for caches [10] is simple to understand, yet offers insight into program behavior.

Multitasking operation systems may interrupt a program at any time and perform a context switch. When a measurement run is interrupted the result can be affected. Techniques to deal with this problem are presented in [8], with a focus on measuring execution time.

In addition, the initial state of the caches can have an influence on the measurement results. It is therefore important to warm or flush the caches. An in detail discussion can be found in [17].

1.3 Contribution of the Thesis

The main contribution of this thesis is of practical nature. We built a tool which simplifies performance measurements and in particular the generation of roofline plots.

The tool provides a set of tested measurement routines, which can be used to measure custom code. Using the provided infrastructure simplifies the analysis of the precision of the generated results.

We presented detailed low level measurement results and analyzed their accuracy and precision. This could be used as reference for future work.

We implemented micro benchmarks to measure the peak performance and memory bandwidth of our measurement system.

We implemented a variety of kernels (in many cases using libraries), generated a roofline plot and discussed them in the context of the roofline model.

Finally, we extended the roofline plots in the following ways:

Series We connected multiple points of a measurement series (for example multiple problem sizes of a single algorithm implementation) with a line.

Series Comparisons When measuring series of different implementations of a single algorithm, we connected equal problem sizes with a dashed line to facilitates the comparison between the implementations.

Error Boxes We repeated our measurements and represented the precision with error boxes within the roofline plots.

Chapter 2

Measurement Setup

In this chapter we first introduce the performance counters, which are used to measure most of the quantities required to generate a roofline plot. Although the performance counters are available on all newer x86 CPUs, they differ between processors. Therefore we describe our measurement machine and how we measured on that particular machine.

Finally, we present our method of cache initialization and how we handle frequency scaling.

2.1 Performance Counters

One of the key features of a CPU is it's speed. But that speed can only be shown if the software running on the CPU is properly optimized. Since optimization often requires detailed understanding of the CPU's internal behavior, the CPU manufacturers decided to directly incorporate performance events and counters into the CPU.

Using special control registers, each counter can be configured to increment when a performance event occurs. Since the counters are implemented in hardware, enabling them does not have any effect on program execution (no overhead).

Recent x86 CPUs typically contain two performance counters and support a large number of performance events, 119 on a CoreDuo. Examples are:

INSTRUCTION_RETIRED occurs for every retired instruction

DBUS_BUSY occurs for every cycle during which data bus is busy

SSE_PRE_MISS occurs when an SSE instruction misses all cache levels

BR_TAKEN_RET occurs for every retired taken branch instruction

When working with the linux operation system, the performance counters are configured and read using a kernel API. The kernel takes care of isolating different processes, such that each process can configure and use the counters according to the specific requirements, without interfering with other processes.

Using the performance counters, it is possible to measure execution time, memory traffic and operation count. For the execution time either the cycle counter or the system timer can be used. The memory traffic can be determined using the performance counters for counting cache misses and write back operations.

An other option is to use the counters for bus transfers. Measuring the operation count depends heavily on the definition of operation, but is typically measurable with the performance counters, too.

2.2 Measurement Machine

All measurements are performed on an IBM X60 Thinkpad, featuring an Intel CoreDuo CPU (Family 6, Model 14). The CPU is based on the Yonah micro architecture, which is similar to the Pentium-M. The CPU contains two cores, each having a 32KB instruction and a 32KB data L1 cache, 8 ways set associative, with 64 bytes line size. The two cores share a 2MB unified L2 cache, again 8 ways set associative and with 64 bytes line size. The core frequency can scale between 1GHz and 1.83GHz. The bus frequency is 167MHz.

The main memory consists of two 2GB DDR2 modules, totaling in 4GB available memory. The theoretical throughput of the memory is 5.12 GB/s, which is 2.80 Bytes per core cycle, if the CPU runs at 1.83GHz.

We used XUbuntu 11.10, running a Linux 3.0.0-16 kernel in 32 bit mode, since the CPU does not support the 64 bit mode. We used GCC 4.6.1.

The performance counters are instrumented using the 'perf event' kernel interface [16], using libpfm4 [2] to generate the required parameters.

We used 'coreduo::UNHALTED_CORE_CYCLES' for measuring time. For the operation count, we used the following definitions for operation:

SinglePrecisionFlop SSE single precision operations.
'coreduo::SSE_COMP_INSTRUCTIONS_RETIRED:SCALAR_SINGLE'
+2*'coreduo::SSE_COMP_INSTRUCTIONS_RETIRED:PACKED_SINGLE'

DoublePrecisionFlop SSE double precision operations.
'coreduo::SSE_COMP_INSTRUCTIONS_RETIRED:SCALAR_DOUBLE'
+2*'coreduo::SSE_COMP_INSTRUCTIONS_RETIRED:PACKED_DOUBLE'

CompInstr Computational instructions retired. Counts SSE instructions and x87 instructions. Used for x87 code.
'coreduo::FP_COMP_INSTR_RET'

SSEFlop SSE operations, sum of SinglePrecisionFlop and DoublePrecisionFlop

We used two variants of measuring the memory transfer volume. The **MemBus** variant uses 64*'coreduo::BUS_TRANS_MEM', which measures the transfers on the system bus. The **MemL2** variant uses the counters for the L2 cache line allocation and eviction, namely 64*('coreduo::L2_LINES_IN:SELF'+'coreduo::L2_M_LINES_OUT:SELF'), combined with 8*'coreduo::SSE_NTSTORES_RET' to take non temporal stores into account.

2.3 Cache

Specially for short running kernels, the initial state of the cache can have a big impact on the memory transfer volume and execution time. The impact can be controlled by making sure the caches are either warm or cold.

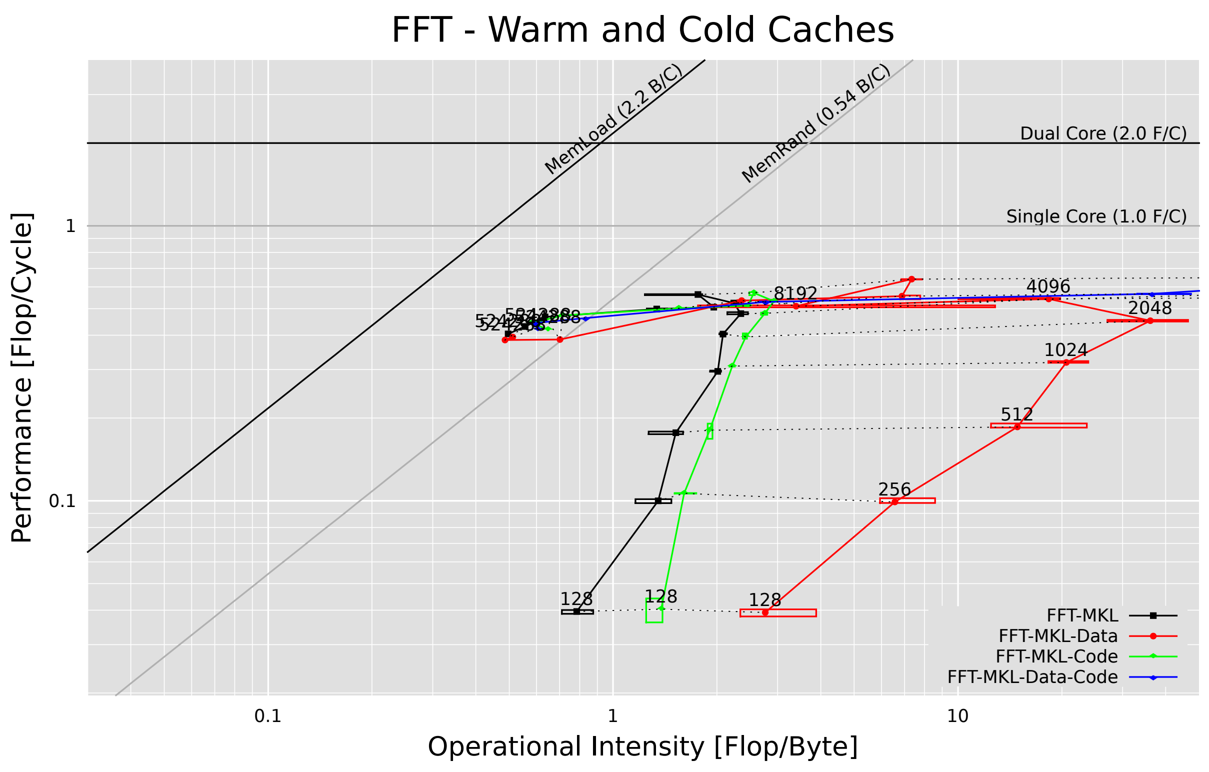


Figure 2.1: Influence of the initial cache state (MemL2)

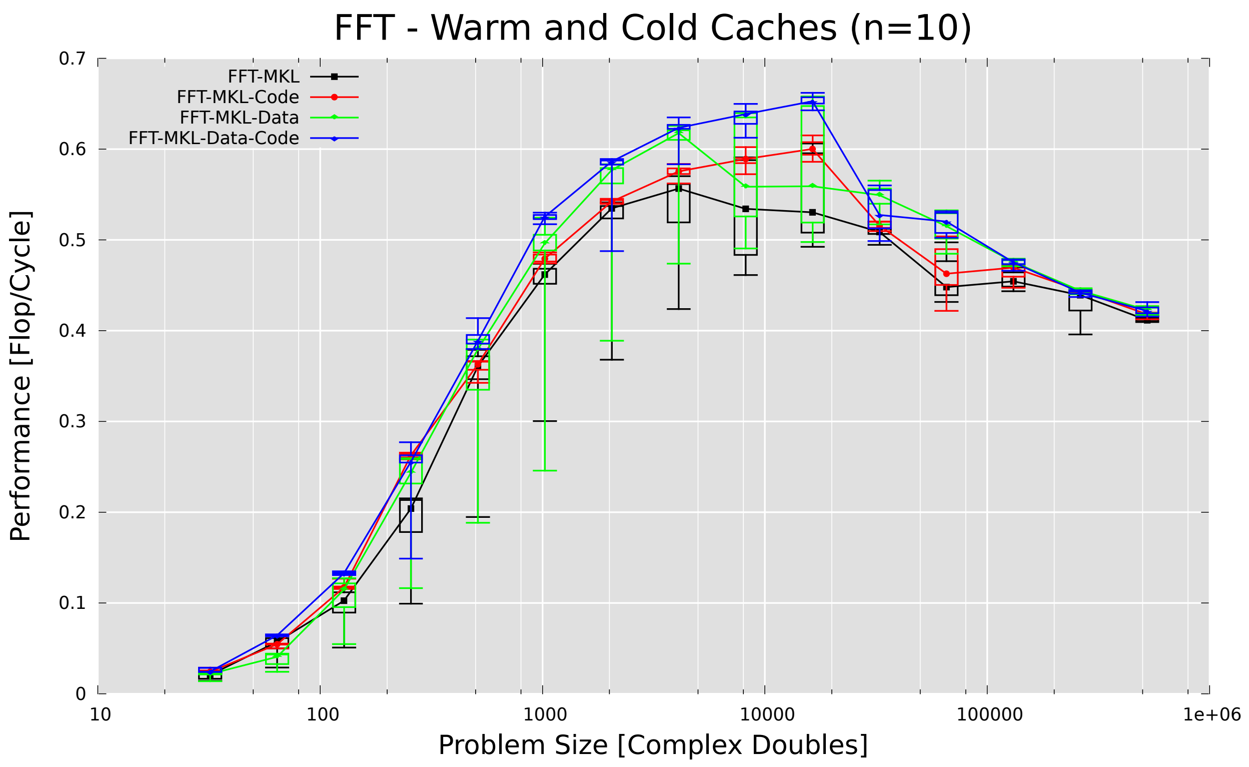


Figure 2.2: Influence of the initial cache state on the Performance

In our tool, it is possible to specify the desired initial state of the code and the data separately. We used the fast fourier transformation implementation of the Intel Math Kernel Libraries as showcase. Figures 2.1 and 2.2 show the results when the cache is cold (FFT-Mkl), only the data is warmed up (FFT-Mkl-Data), only the code is warmed up (FFT-Mkl-Code) or both data and code is warmed up (FFT-Mkl-Data-Code).

In the roofline plot (fig. 2.1) we repeated each measurement 10 times and showed the median result, as well as the 25% and 75% percentile as a box around the median point. We connected the median points for the same problem size with dashed lines.

If only the code cache is warmed up, the difference to completely cold caches is small. The effect of warming up the data caches more significant.

If the data and the code is loaded into the caches initially, the operational intensity increases again compared to only loading the data. Short before the cache size is exceeded by the size of the data and the code, the operational intensity becomes very large.

For large problem sizes, the initial state of the cache has a negligible impact.

To get a better impression on the effects on performance, figure 2.2 shows the problem size on the x axis and the performance on the y axis. For a problem size of 10’000, the performance difference between cold and warm caches (code and data) is around 20%.

The following snippet shows the relevant logic for initializing the caches:

```
// should the code cache be warm?
if (getWarmCode()) {
    // Tell the kernel to warm the code cache,
    // which usually results in the kernel
    // beeing executed once.
    getKernel()->warmCodeCache();

    // Should we clear the data?
    if (!getWarmData()) {
        getKernel()->flushBuffers();
    }
} else { // Code cache should be cold.
    // Access a large memory buffer and execute
    // a lot of code, this clears the code cache.
    clearCaches();

    // Should the data be warm?
    if (getWarmData()) {
        // Warm the data cache by accessing each
        // cache line of the data buffer(s).
        getKernel()->warmDataCache();
    } else {
        // The data cache should be cold.
        getKernel()->flushBuffers();
    }
}
```

Getting cold caches can be quite tricky. See [17] section '3. CACHE FLUSHING METHODS WHEN TIMING ONE INVOCATION' for details.

To clear the code cache, we use the traditional method of accessing a large memory buffer and executing more than 32KB code to flush the L1 code cache.

In addition, the kernels report the buffers they allocated. This is used to flush these buffers with the 'clflush' instruction, not affecting code which is already in the cache. This is implemented by the flushBuffers() method in the above code snippet.

2.4 Frequency Scaling

The core frequency is not constant in current processors. Since the memory latencies and throughputs do not scale with the core frequency, a lower core frequency will generally cause a higher percentage of the peak performance to be reached by the kernel.

On Linux, frequency scaling is controlled so called governors. We used the 'performance' governor for all our measurements, which fixes the core frequency to the maximum.

Chapter 3

Single Threaded Accuracy and Precision

In this chapter we analyze the quality of our measurement results for single threaded workloads.

First, let us recapitulate the definitions of accuracy and precision. [18] Accuracy of a measurement system is the closeness of the measurement results to the true value. Precision is the degree to which repeated measurements show the same result.

A standard technique to increase precision is to repeat a measurement and use a measure of tendency. This generally increases the precision of the result, but does not improve the accuracy in presence of systematic errors.

To analyze the results of single threaded measurements we designed three groups of measurements. Each group is focused on one specific quantity, namely time, transfer volume and operation count. We measured on an idle system. All measurements are executed with cold caches.

In all groups we used the following kernels:

ADD Repeat adding a constant to an accumulator. Everything is performed in registers. We use multiple accumulators and unroll the loop.

MUL Repeat multiplying an accumulator with a constant. Everything is performed in registers. We use multiple accumulators and unroll the loop.

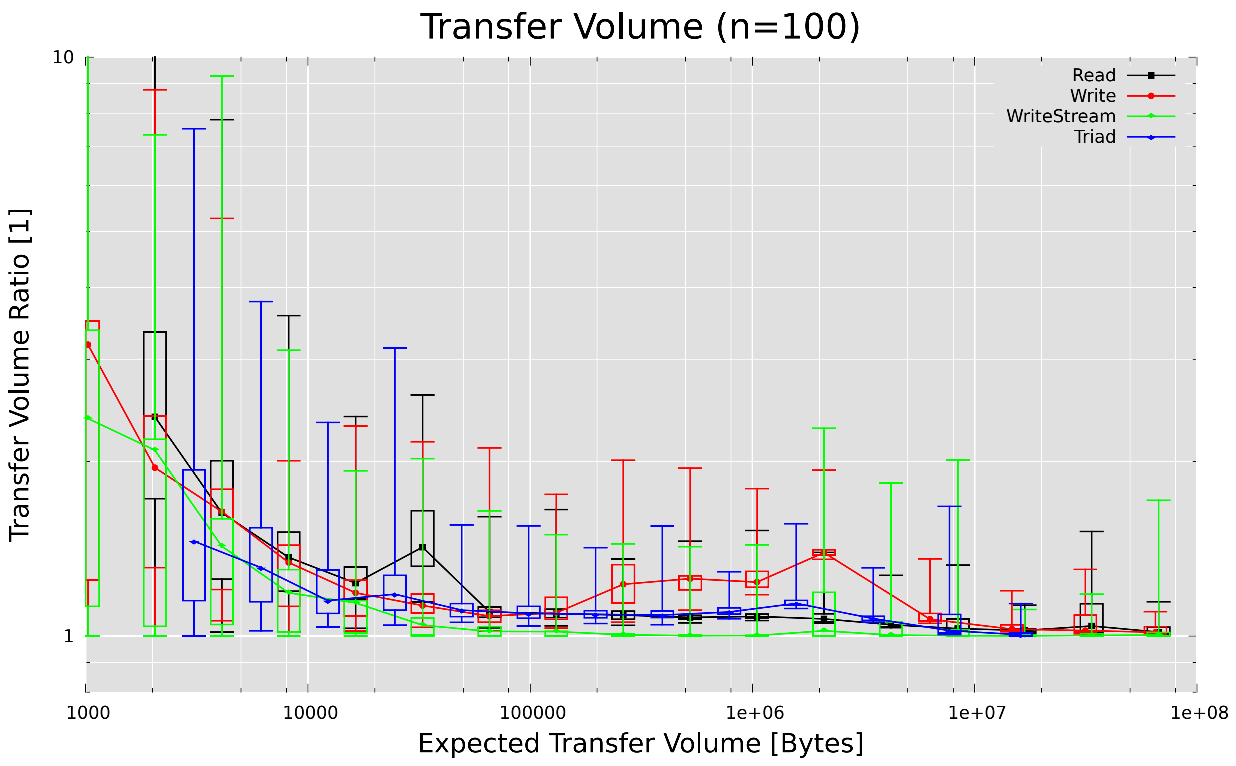
Read Read a buffer from memory

Write Overwrite a buffer in memory

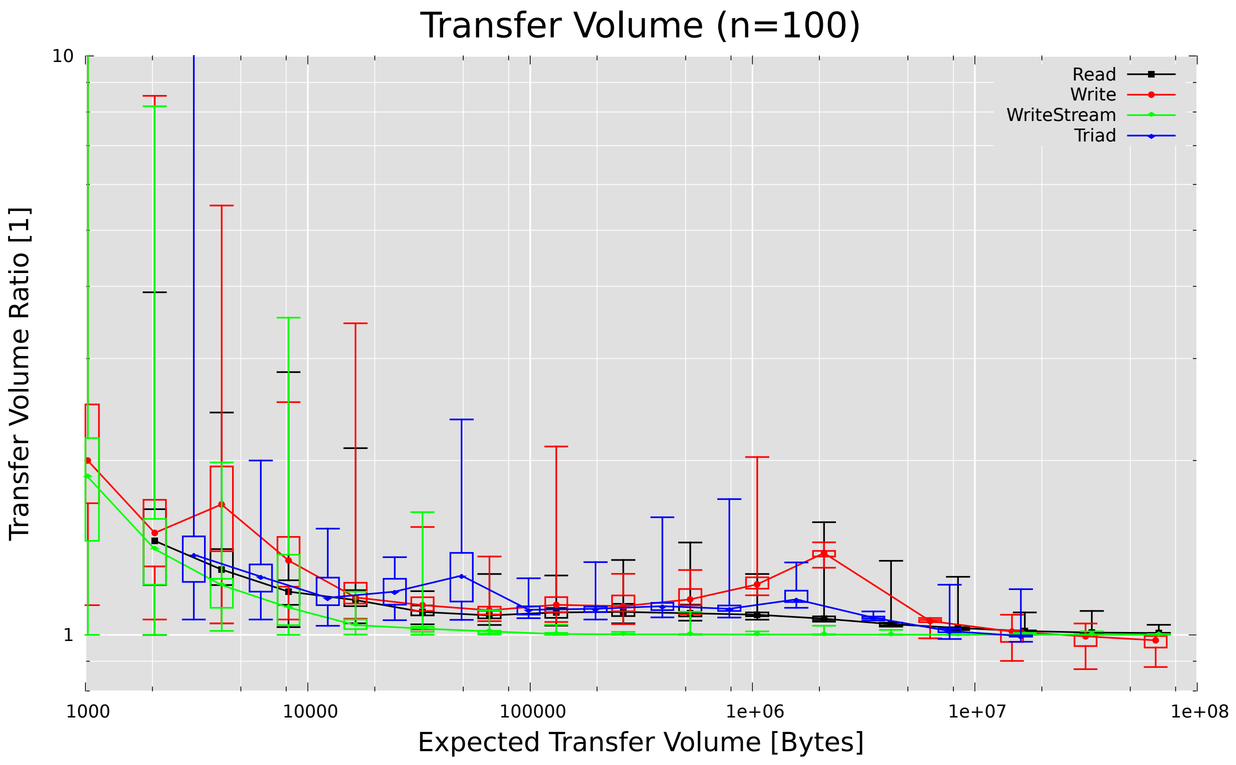
WriteStream Overwrite a buffer in memory, using non temporal store instructions

Triad Perform $a_i = b_i + k * c_i$. This involves reading a, b and c and writing a back.

Read, write and WriteStream read or write a single memory buffer of size S. The Triad kernel operates on three buffers of size S.



(a) MemBus



(b) MemL2

Figure 3.1: Ratio of the Actual Transfer Volume to the Expected Transfer Volume

3.1 Transfer Volume

To evaluate the accuracy of our results, we tried to estimate the amount of memory Θ that has to be transferred. For pure reading, we expect to observe the whole buffer being transferred to the core ($\Theta = S$). When writing is involved, we expect some of the writes to be held back in the cache. For the write kernel the buffer is loaded into the CPU before beeing overwritten. In addition we presume that the whole cache is used for write back caching. Thus the estimated transfer volume is $\Theta = S + \max(0, S - 2MB)$.

For the triad kernel, all three buffers have to be loaded into the CPU and one has to be written back. We presume that one third (due to the three buffers) of the cache is used for write back caching. ($\Theta = 3S + \max(0, S - \frac{2}{3}MB)$) The streaming write kernel should write the whole buffer once ($\Theta = S$).

In figure 3.1 we show the expected transfer volume Θ on the x axis and the ratio of the actual and the expected transfer volume on the y axis ($\frac{\tau}{\Theta}$). We measured system bus transfers in sub figure **a** (MemBus) and L2 cache line allocations in sub figure **b** (MemL2).

The distribution of the results is shown using box plots. [19] For each expected transfer volume, the measurement is repeated $n = 100$ times. The median is shown with a point and connected to the medians of the other transfer volumes with a line. Around the point a box is plotted. The top of the box represents the 75th percentile, the bottom the 25th percentile. The ends of the whiskers represent the minimum and maximum of the observed results.

With our measurement setup, we roughly observe the expected transfer volumes. Around an expected transfer volume of 2MB we overestimate the influence of the write back cache. The CPU seems to write data back to the memory before the cache is completely filled.

The MemBus and MemL2 variants produce similar result. Note that the MemL2 variant measures less than the expected memory transfer for large buffer sizes.

To show the error of the measurement results we defined the err(x) function:

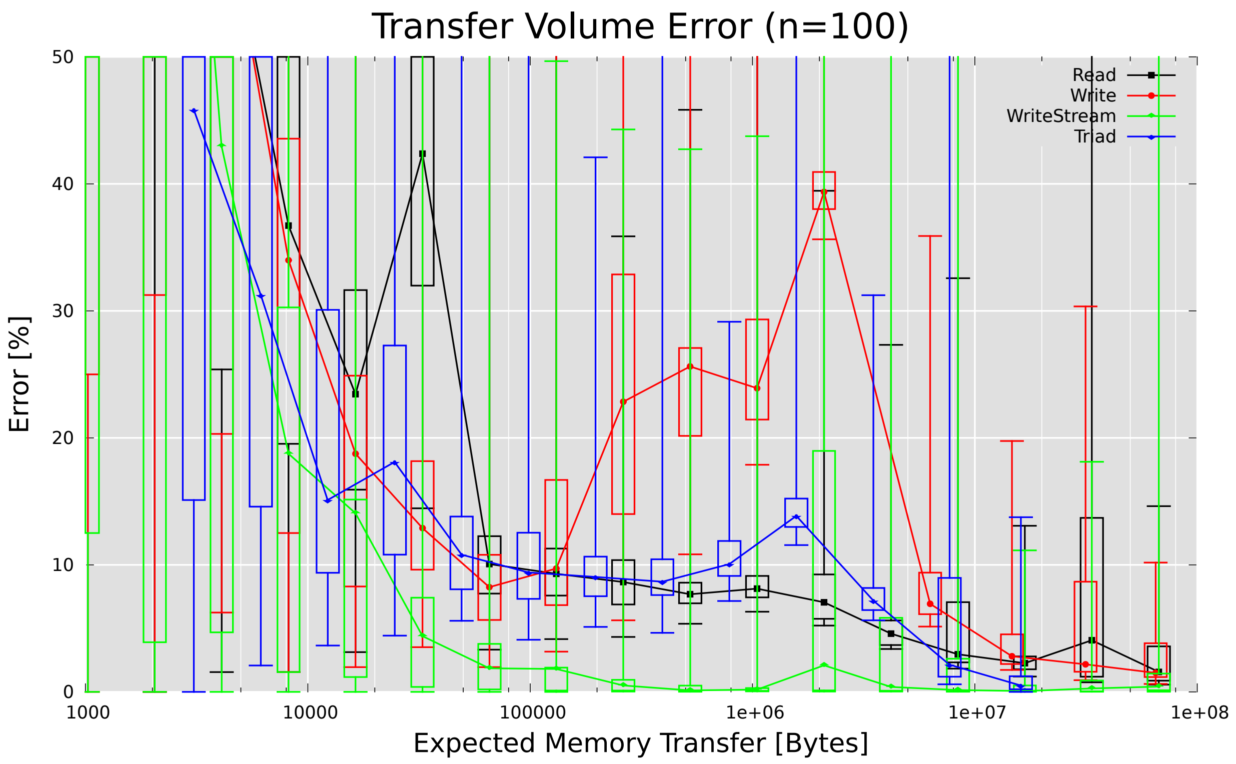
$$\operatorname{err}(x) = \begin{cases} x > 1 & (x-1) * 100\% \\ x \leq 1 & (\frac{1}{x} - 1) * 100\% \end{cases}$$

We used the err(x) function for the y axis of figure (3.2). The spread of the MemL2 variant is better than the spread of the MemBus variant, while the offset is comparable.

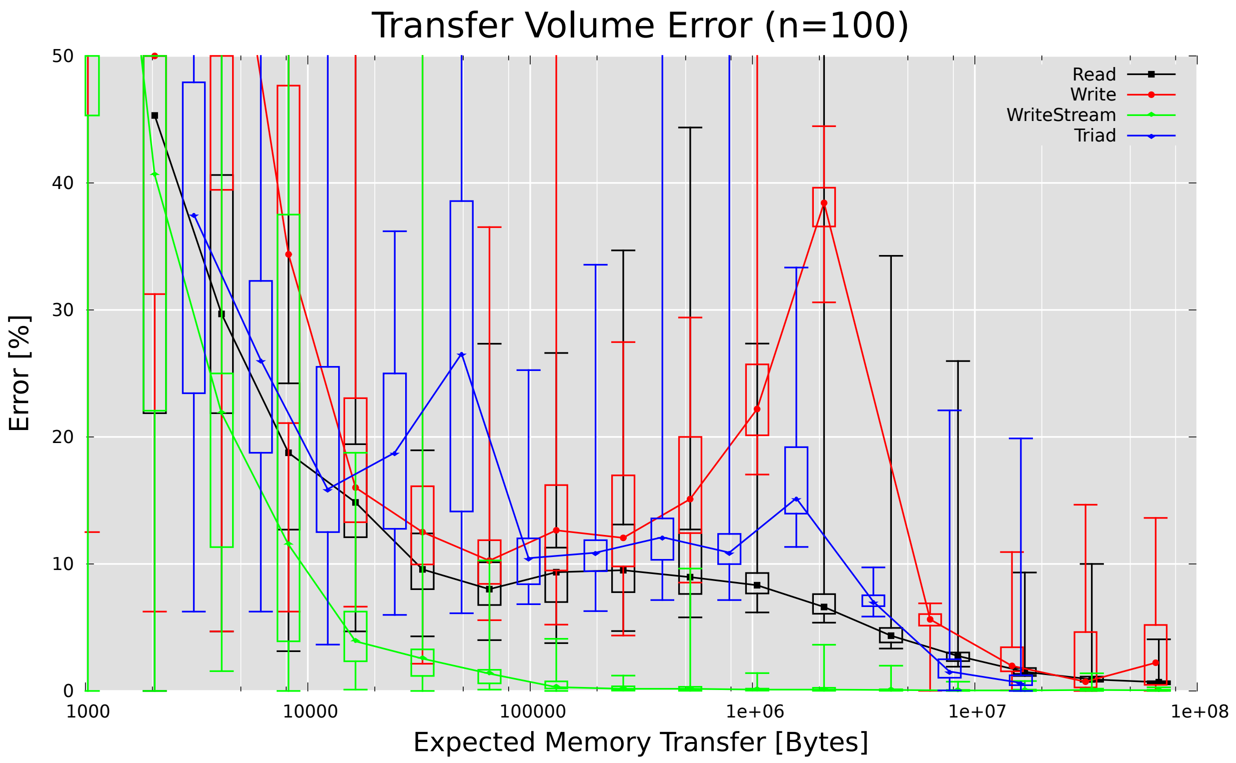
To validate our assumption on the write back cache, we measured the memory transfer while flushing the cache using ‘CLFLUSH’ after the execution of the kernel completes. (fig. 3.3)

There is a small overhead for the read kernel. For the write and the triad kernel, all written data is transferred while flushing the caches as long as the data fits into the cache.

If the data does not fit into the cache anymore, almost the whole 2MB 2nd level cache is used for write back caching in case of the write kernel. For the Triad kernel one could suspect that each buffer receives about one third of the cache, which is affirmed by the data.



(a) MemBus



(b) MemL2

Figure 3.2: Error of the Transfer Volume: $\mathrm{err}(\frac{\tau}{\Theta})$

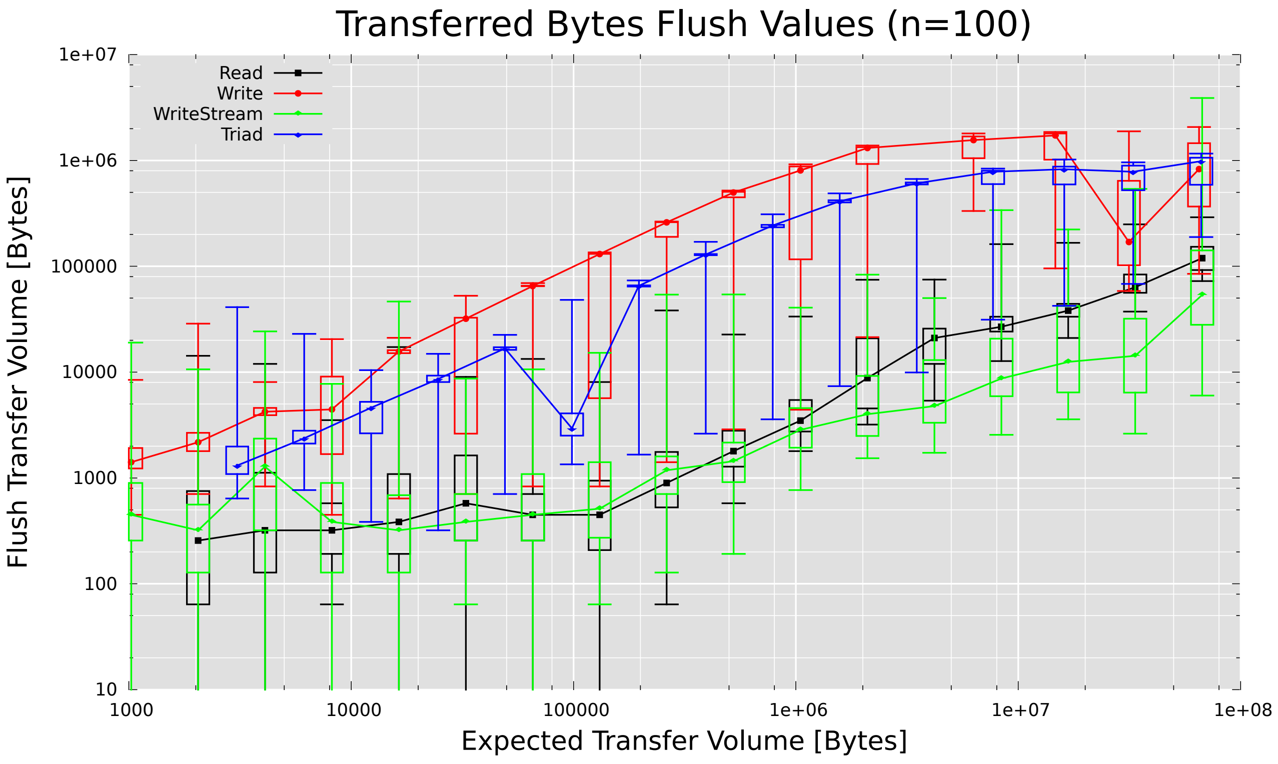
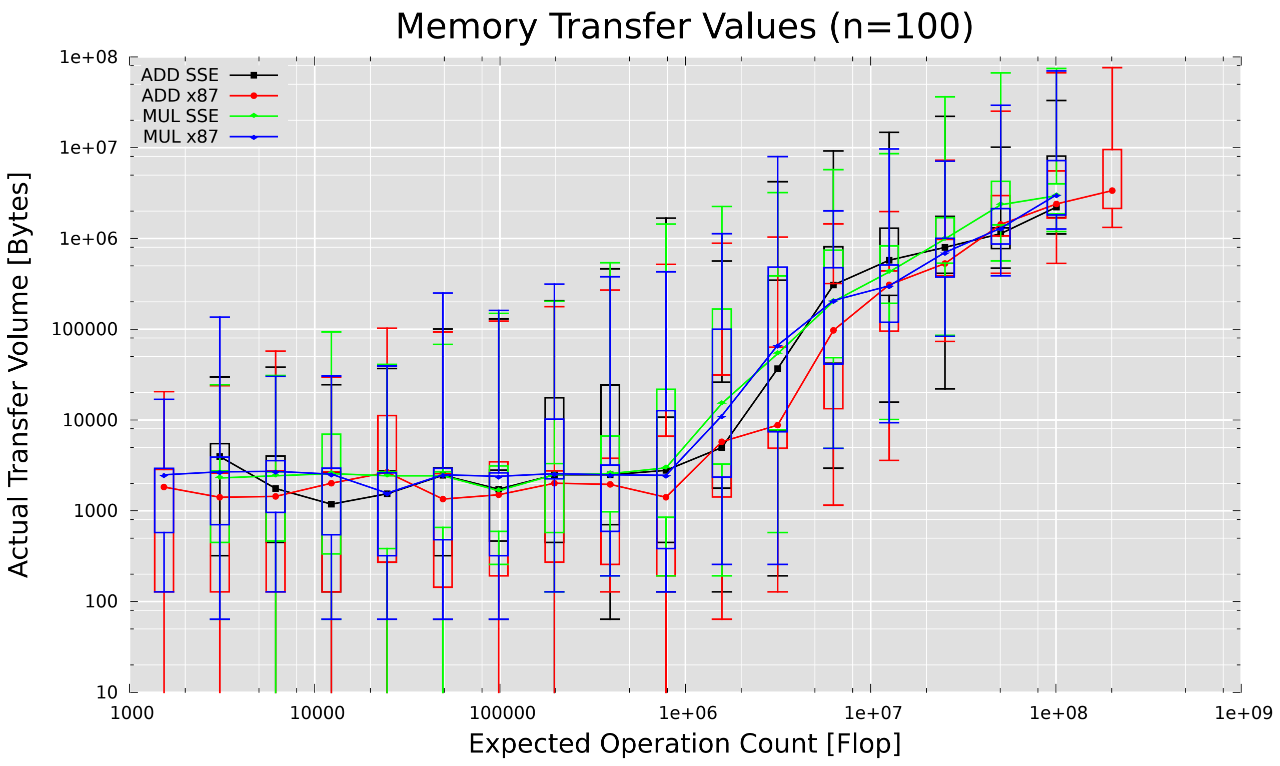


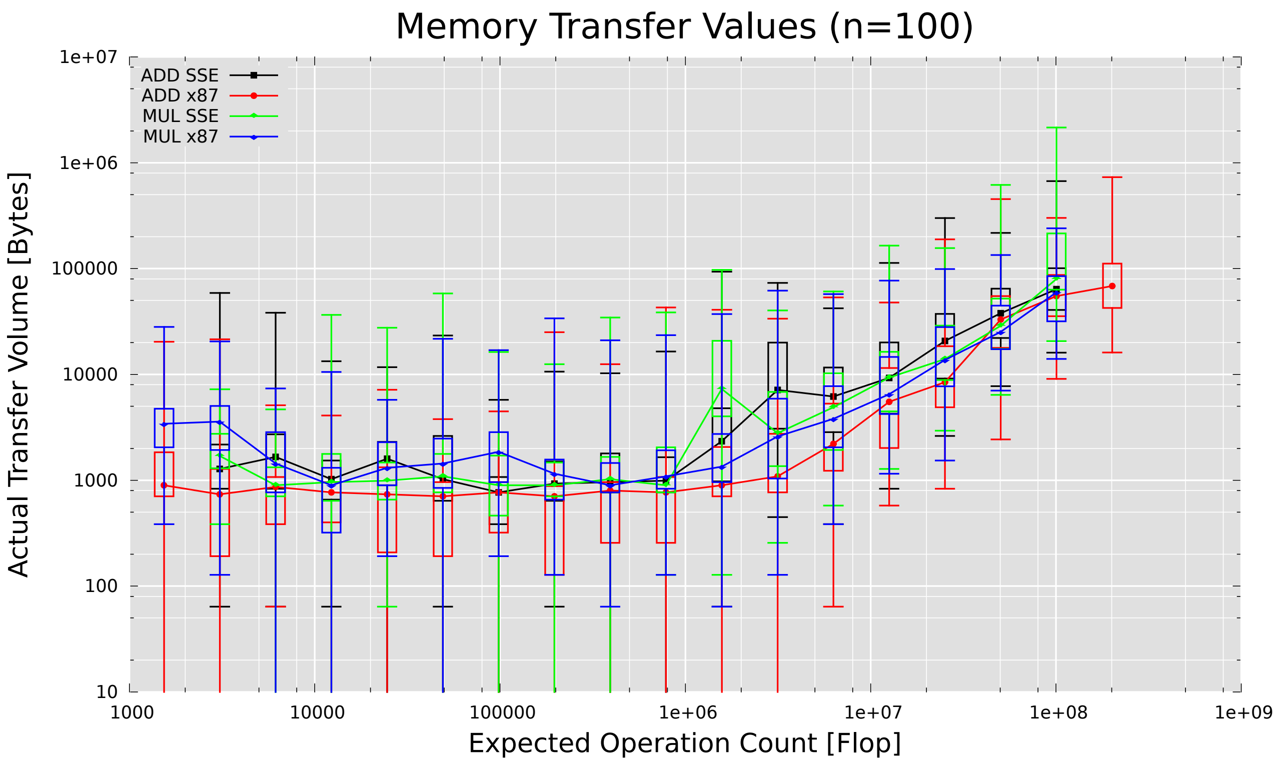
Figure 3.3: MemL2: Memory Transfer during Cache Flush after kernel execution

When measuring on an Intel Core, we observed a strange behavior. While the Read and the Triad kernels behaved as expected, the we observed a memory transfer of half the buffer size for the write kernel, independent of the buffer size. We used the SSE intrinsics. Switching to the normal integer registers produced the expected results.

When measuring the memory transfer of the arithmetic kernels, we observe an overhead of around 2.5KB for for short running kernels. When the execution time exceeds 1ms to 10ms, the memory transfer starts to correlate with the execution time. There is a huge difference between the BusMem and BusL2 variants. The BusMem variant measures an overhead of around one byte every 20 cycles, while the BusL2 variant measures one byte every 300-1600 cycles (see fig. 3.4 and 3.5) This difference is plausible since the BusMem variant is likely to pick up any noise on the memory bus.



(a) MemBus



(b) MemL2

Figure 3.4: Transferred Bytes of the Arithmetic Kernels

3.2 Time

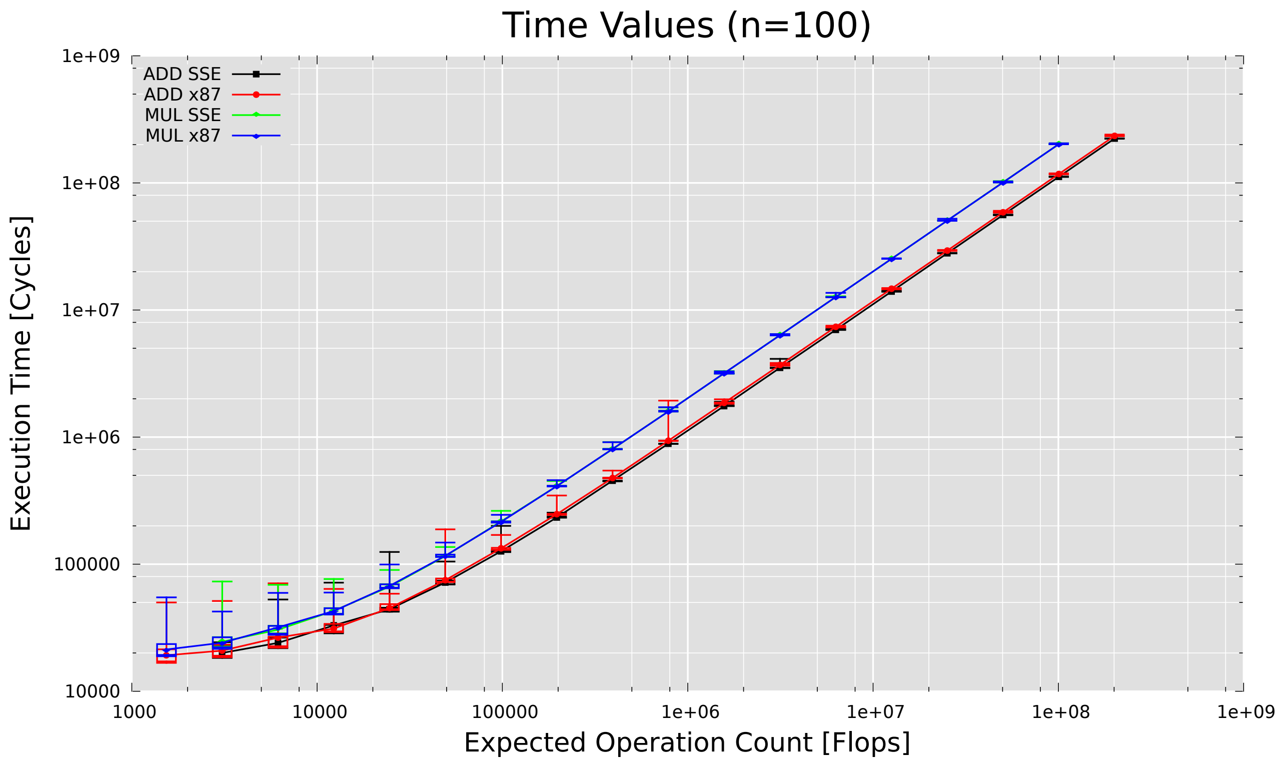


Figure 3.5: Execution Time of the Arithmetic Kernels

We analyzed the execution time of the arithmetic ADD and MUL kernels. Both should theoretically cause no memory transfer. Figure 3.5 shows the execution time of the kernels for different iteration counts. The execution time is one cycle per operation for the ADD kernel and two cycles per operation for the MUL kernel. The difference between using the legacy x87 floating point instructions and SSE is negligible. These results are consistent with the information found in the Intel manual.

For the error plot we use the minimal execution time for each expected operation count as reference. The error is small for operation counts above 1'000'000 (fig 3.6).

For the kernels causing memory transfer, we observe a strong correlation between expected memory transfer and execution time (fig. 3.7), but the errors far exceed those observed for the arithmetic kernels (fig. 3.8).

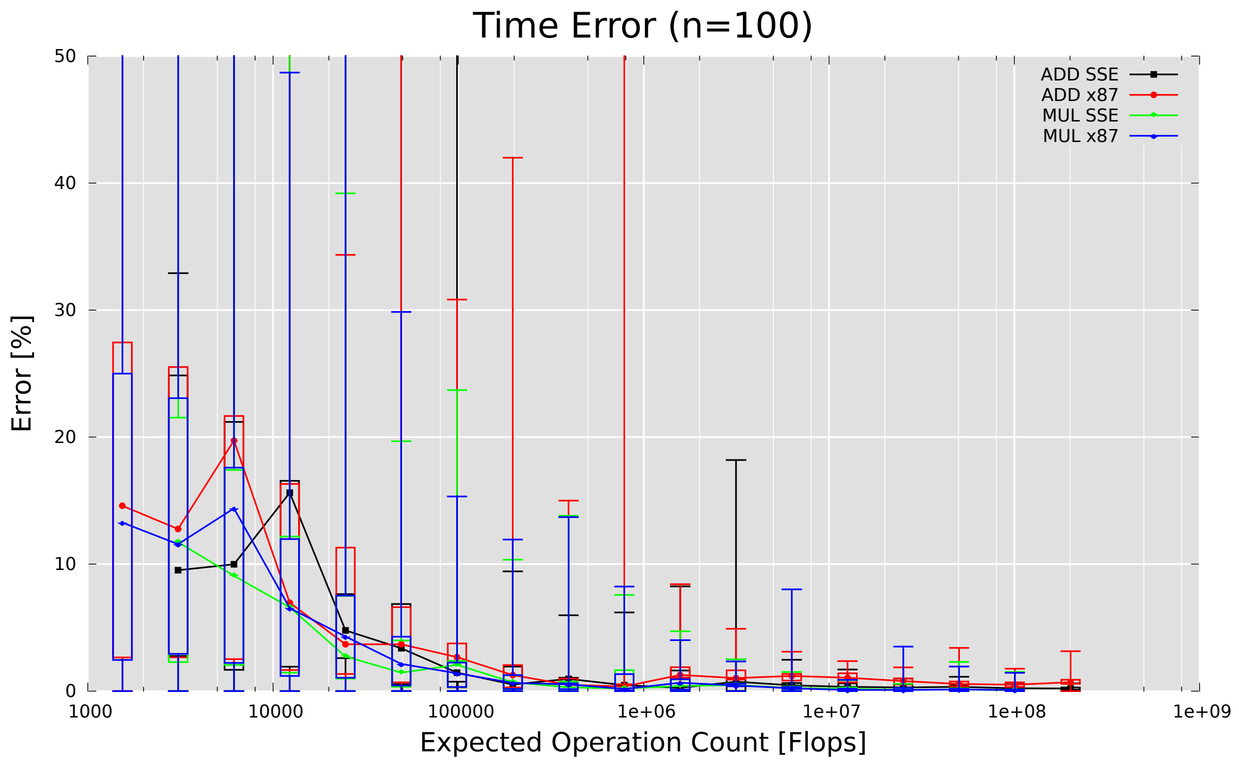


Figure 3.6: Error of the Execution Time of the Arithmetic Kernels: $\text{err}\left(\frac{\text{Execution Time}}{\text{min(Execution Time)}}\right)$

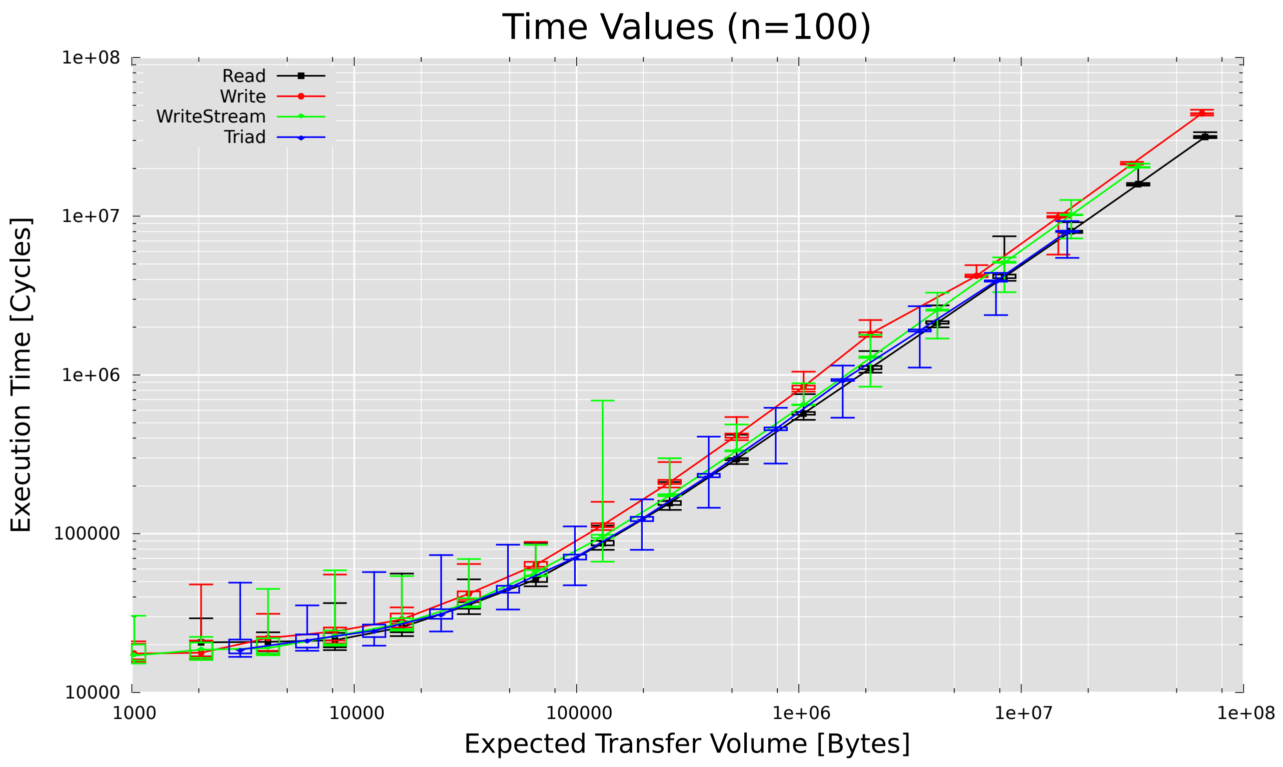


Figure 3.7: Execution Time of the memory kernels

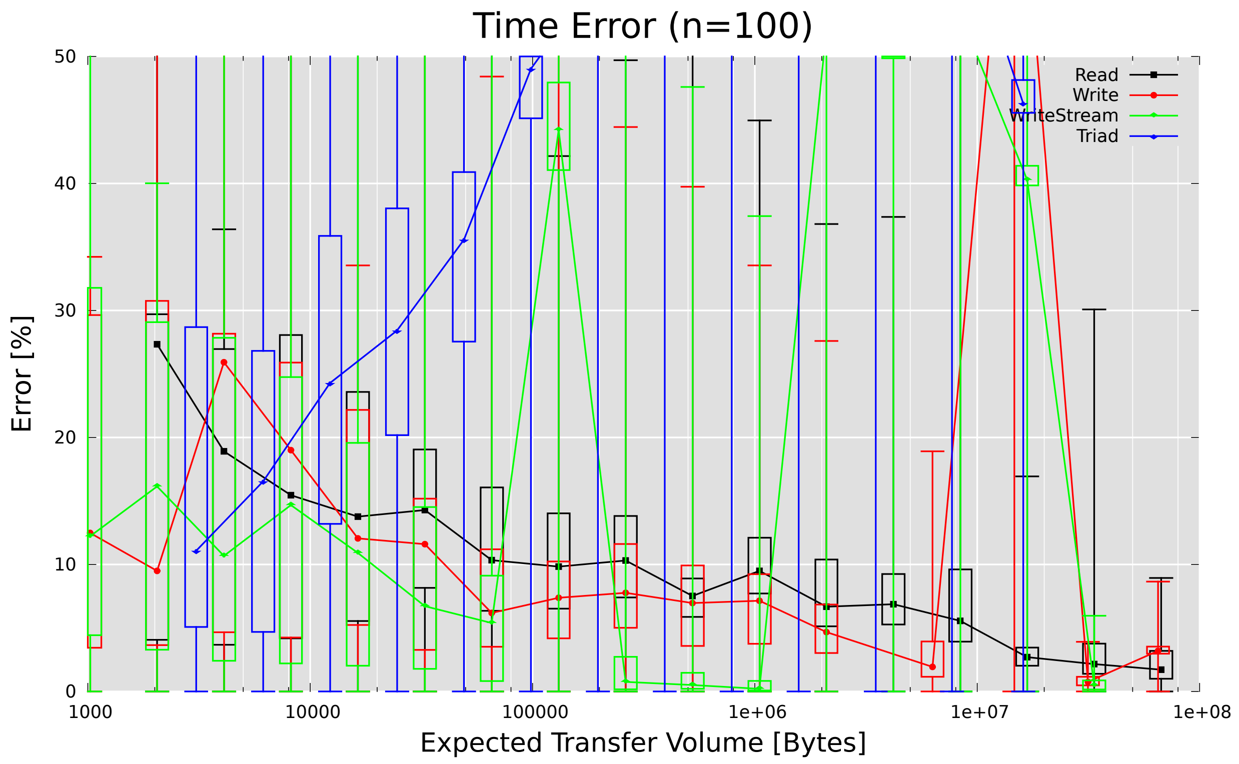


Figure 3.8: Error of the Execution Time of the Memory Intensive Kernels: $\text{err}\left(\frac{\text{Execution Time}}{\min(\text{Execution Time})}\right)$

3.3 Operation Count

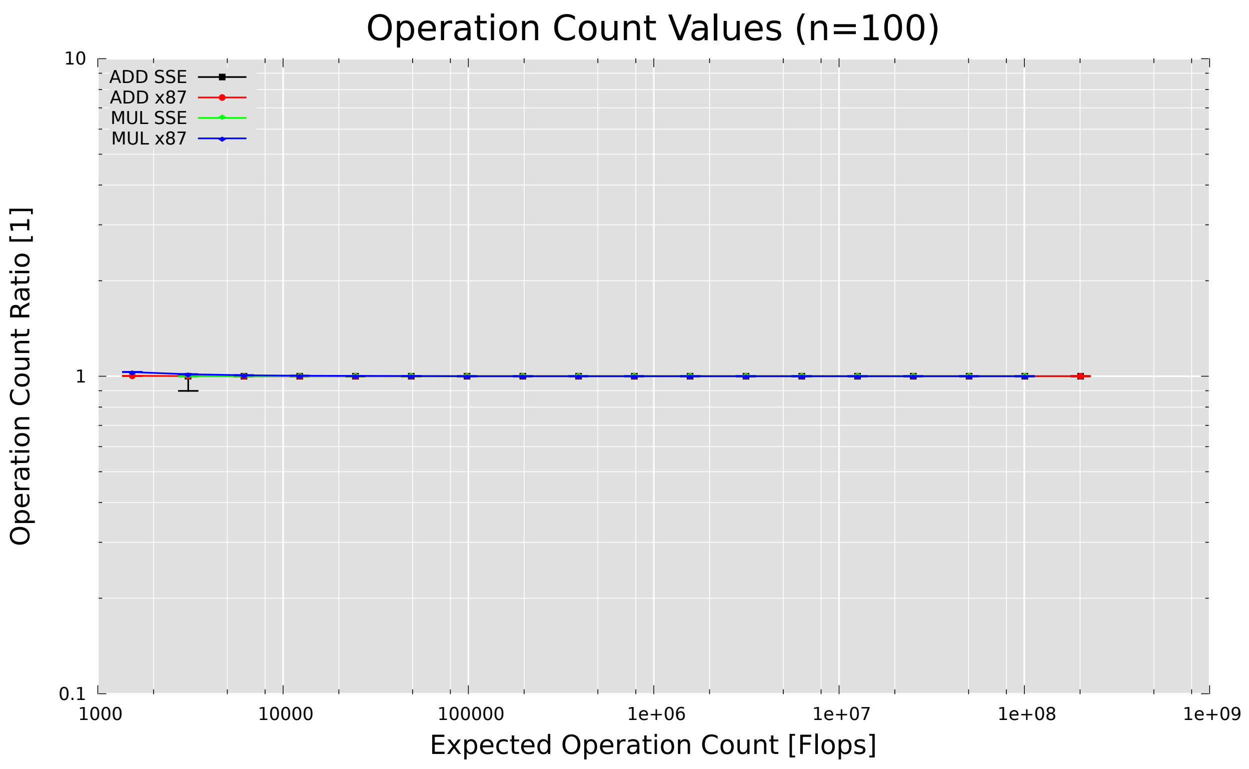


Figure 3.9: Operation Count: $\frac{\text{Actual Operation Count}}{\text{Expected Operation Count}}$

Figure 3.9 shows that we get almost perfect results for measuring the operation count.

3.4 Handling the Variance

In the previous sections we saw that our measurements produce plausible results. As soon as the kernels cause memory transfer, we observe a high variance. We presume this is due to various activities happening in the background during the measurement. (Task switches etc).

These activities cause events to be measured which are not related to the execution of the kernel. Fortunately, additional events only increase the measured execution time, memory transfer or operation count. Ideally, at least one execution does not include any overhead. In this case the true value is the minimum value observed.

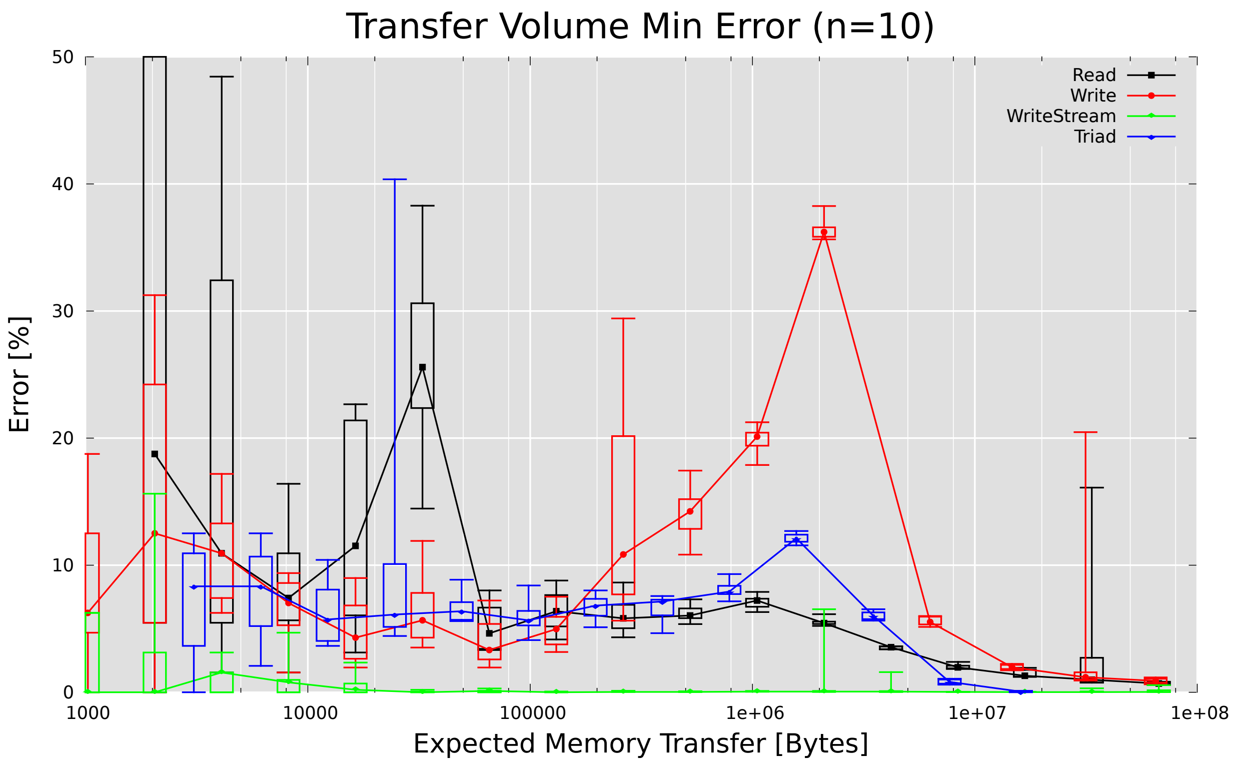
Based on these considerations, we repeat each measurement $K = 10$ times and discard all but the minimum value. Results generated using this scheme are marked by adding value of K as subscript.

Using this scheme for measuring the transfer volumes, the samples fall in a range of about 5%. The bias is under 10% except for transfer volumes under 20KB and with a peak of up to 40% around 2MB for the kernels involving writes. The differece in precision between MemBus and MemL2 disappeared almost completely. For large transfer volumes, the accuracy of the MemBus variant is superior. (fig. 3.10)

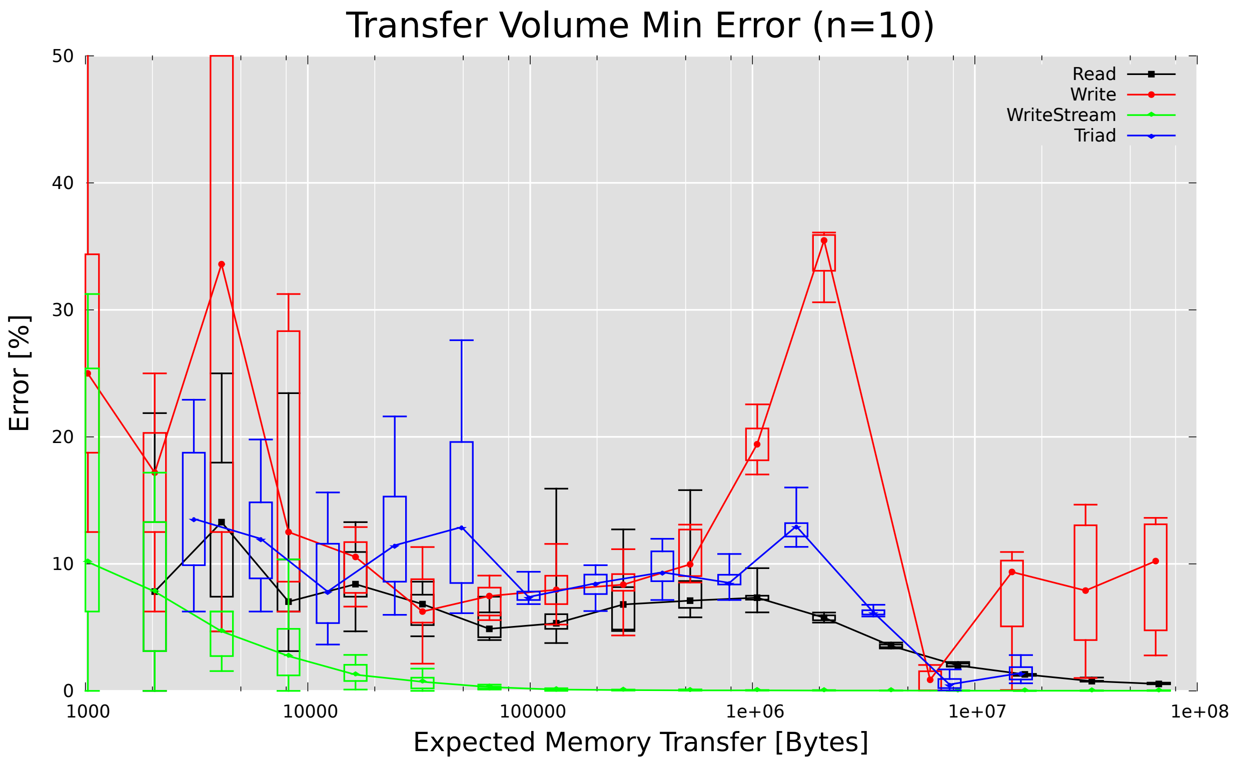
Applying the same scheme to measuring the execution time of the memory kernels yields a precision of about 10%. Since we cannot derive an expected value, we chose the minimum observed value as reference. (fig. 3.11)

Finally, figures 3.12 and 3.13 show that the arithmetic kernels do not pose any problems.

Summarizing our results, using the minimum of K scheme with $K = 10$ the arithmetic kernels can be measured with high precision and accuracy. The results of measuring execution time and memory transfer of memory intensive kernels are scattered within 10% of the reference value. The memory transfer volume is overestimated by about 10% for transfer volumes under 10MB.



(a) MemBus



(b) MemL2

Figure 3.10: Error of the Transfer Volume with $K = 10$: $\text{err}\left(\frac{\tau_{10}}{\Theta}\right)$

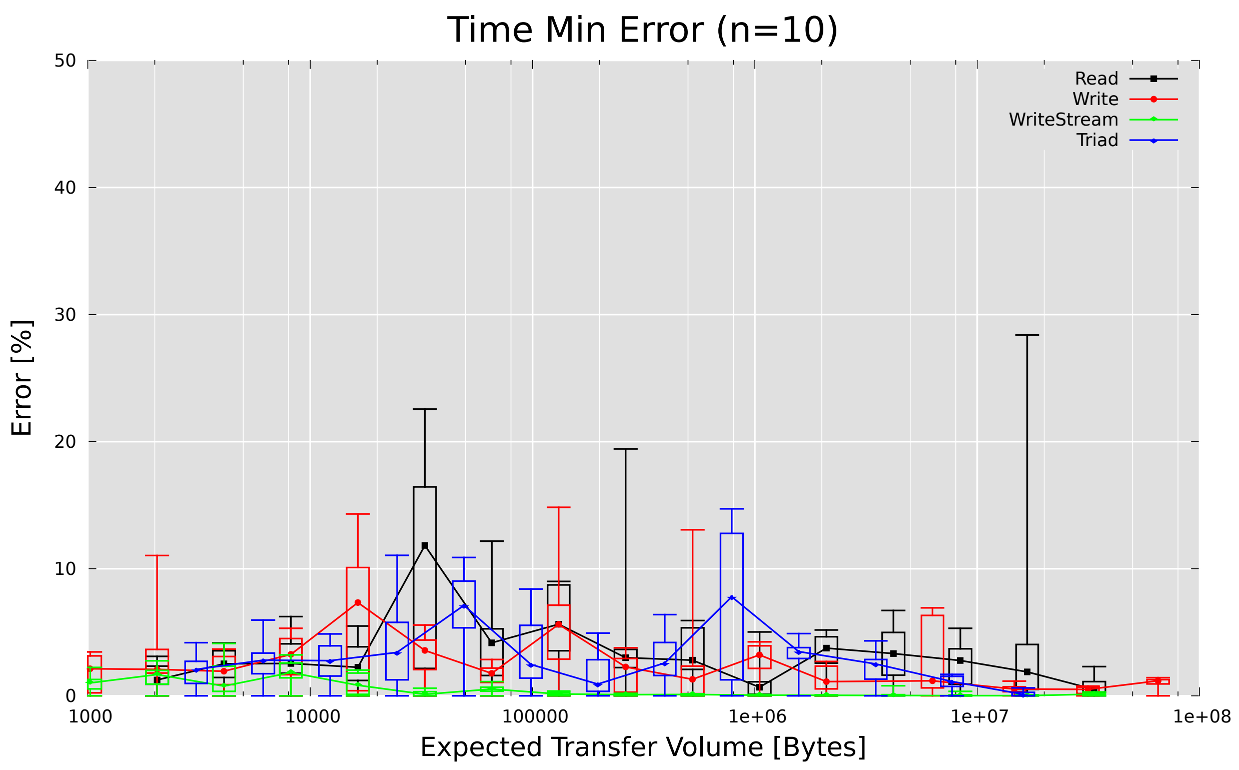


Figure 3.11: Error of the Execution Time with $K = 10$: err $\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

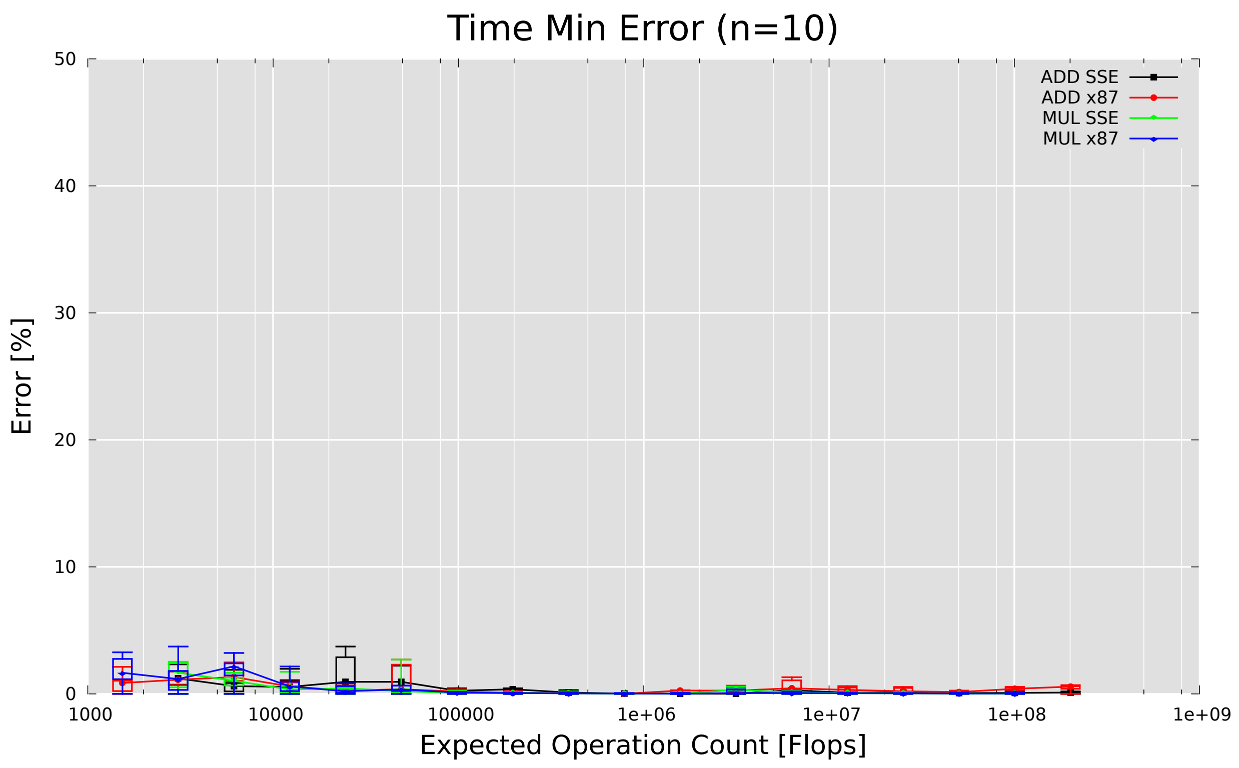


Figure 3.12: Error of Execution Time with $K = 10$: err $\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

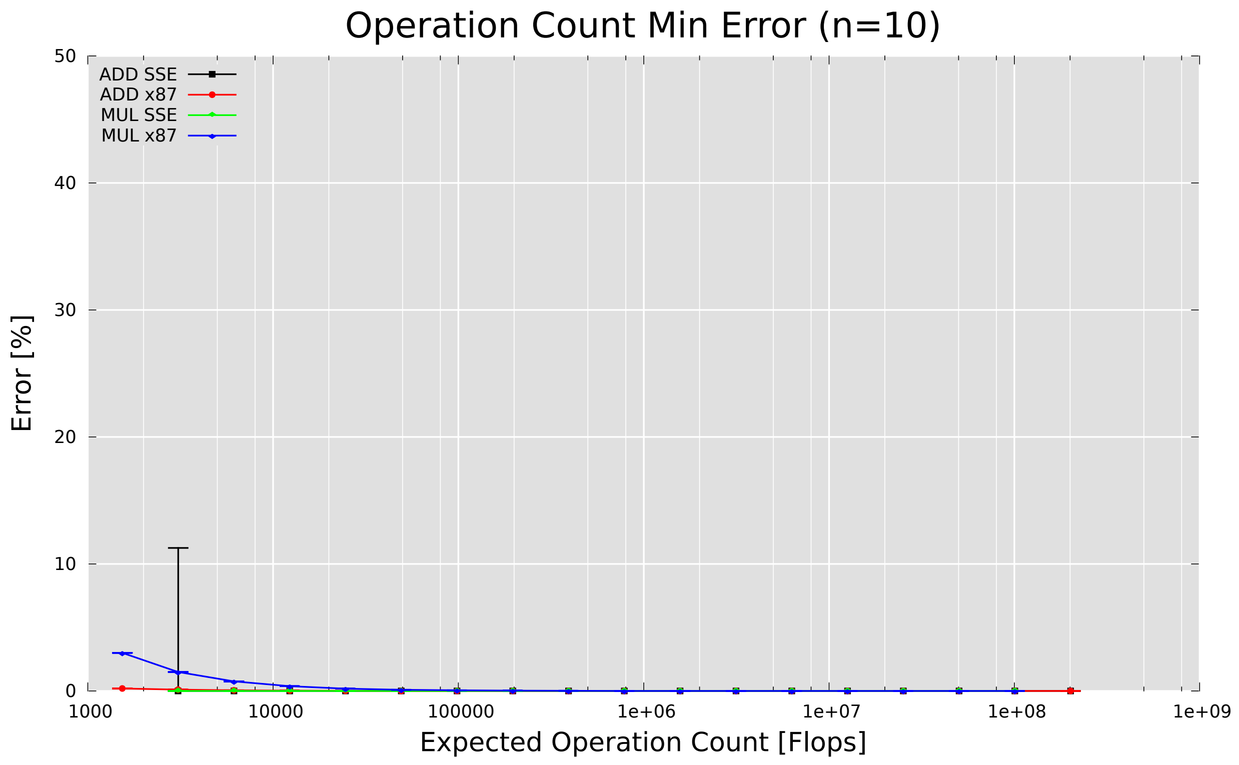


Figure 3.13: Error of Operation Count with $K = 10$: $\text{err}(\frac{\text{Actual Operation Count}_{10}}{\text{Expected Operation Count}})$

Chapter 4

Multi Threaded Accuracy and Precision

For the validation of multi threaded measurements, we use the same measurements as for the single threaded validation, but run a separate workload instance on each core. Using a barrier, we make sure the two kernels start at the same time.

Since the Yonah architecture does not utilize HyperThreading, we do not expect an arithmetic kernel running on one core affecting a memory intensive kernel on the other core. Therefore the workloads run in parallel are not mixed.

4.1 Transferred Volume

The MemBus variant measures all transfers on the bus. Therefore, the measurer of each workload should see the traffic of both kernels. The MemL2 variant should separate the traffic of the two cores. On figures 4.1, 4.2, 4.3 and 4.4 we see the expected results for the MemL2 variant. The precision is comparable to the single threaded measurements, the overhead is slightly higher.

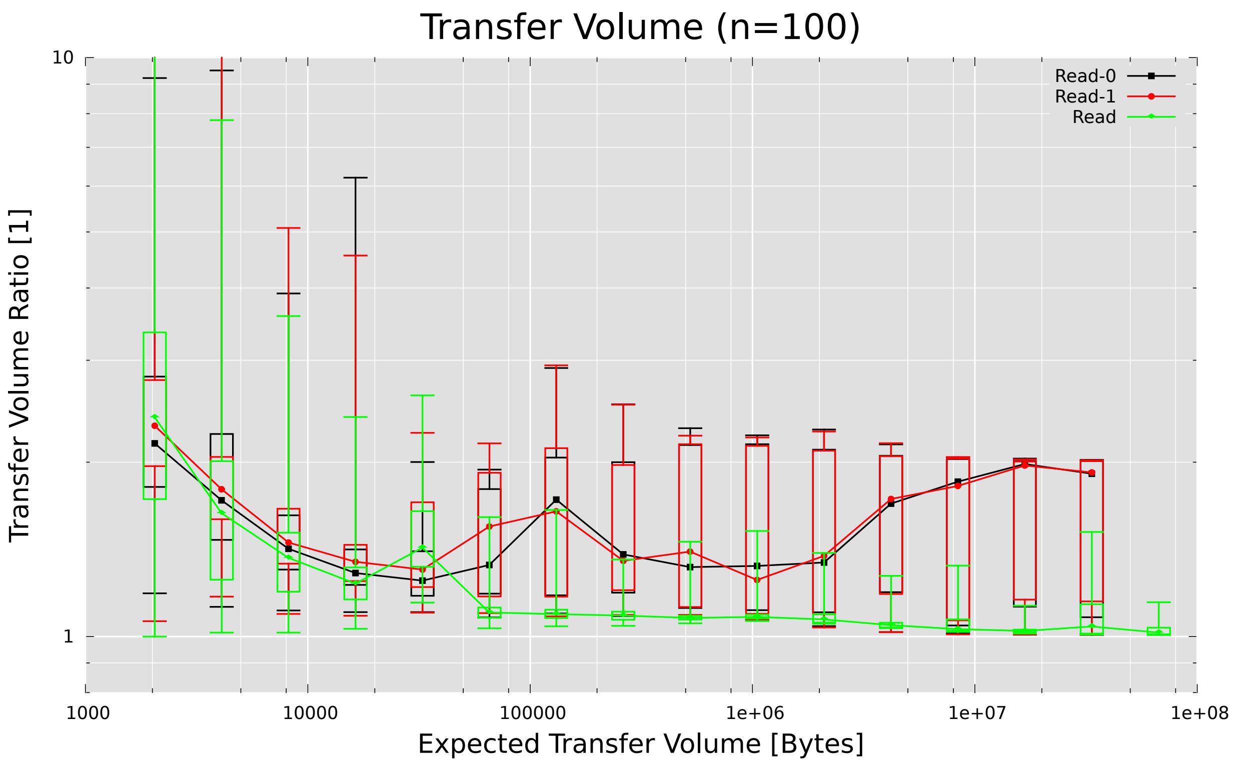
In contrast, the MemBus variant shows a huge variation. Choosing 4MB as representative buffer size, figures 4.5, 4.6, 4.7 and 4.8 show the distribution of the results. There is a cluster around the expected result of twice the expected transfer volume (each workload seeing the transfer of both cores), but also a cluster around once the expected transfer volume. It seems that in about half of the measurement runs, the kernel only see their own traffic.

4.2 Time

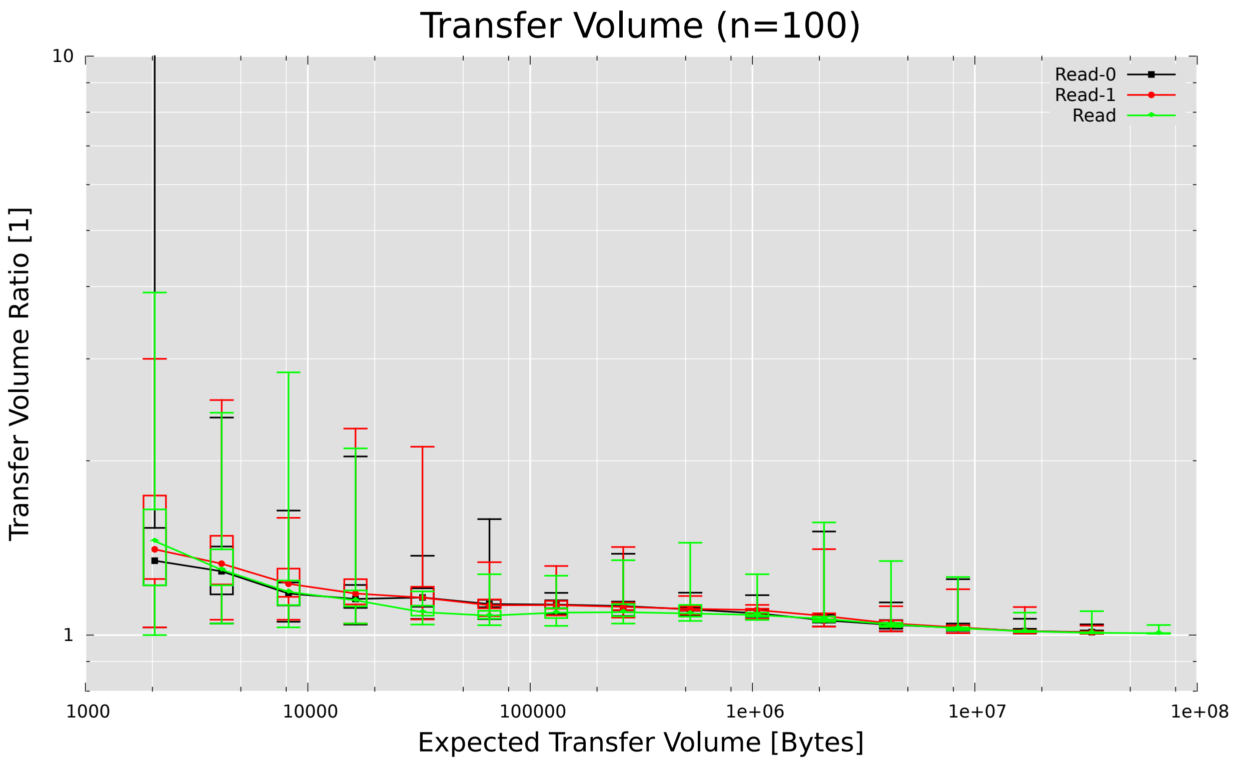
As expected, running two arithmetic kernels in parallel does not affect the measured execution time (fig. 4.9).

For the memory kernels we do not see much difference in execution time between the single threaded and the multi threaded results for expected transfer volumes below 1MB. We expected to observe a larger difference. For larger buffers the expected difference of about a factor of two can be observed, although outliers are frequent. (fig. 4.10 and 4.11)

For the triad kernel there is almost no difference. (fig. 4.12)

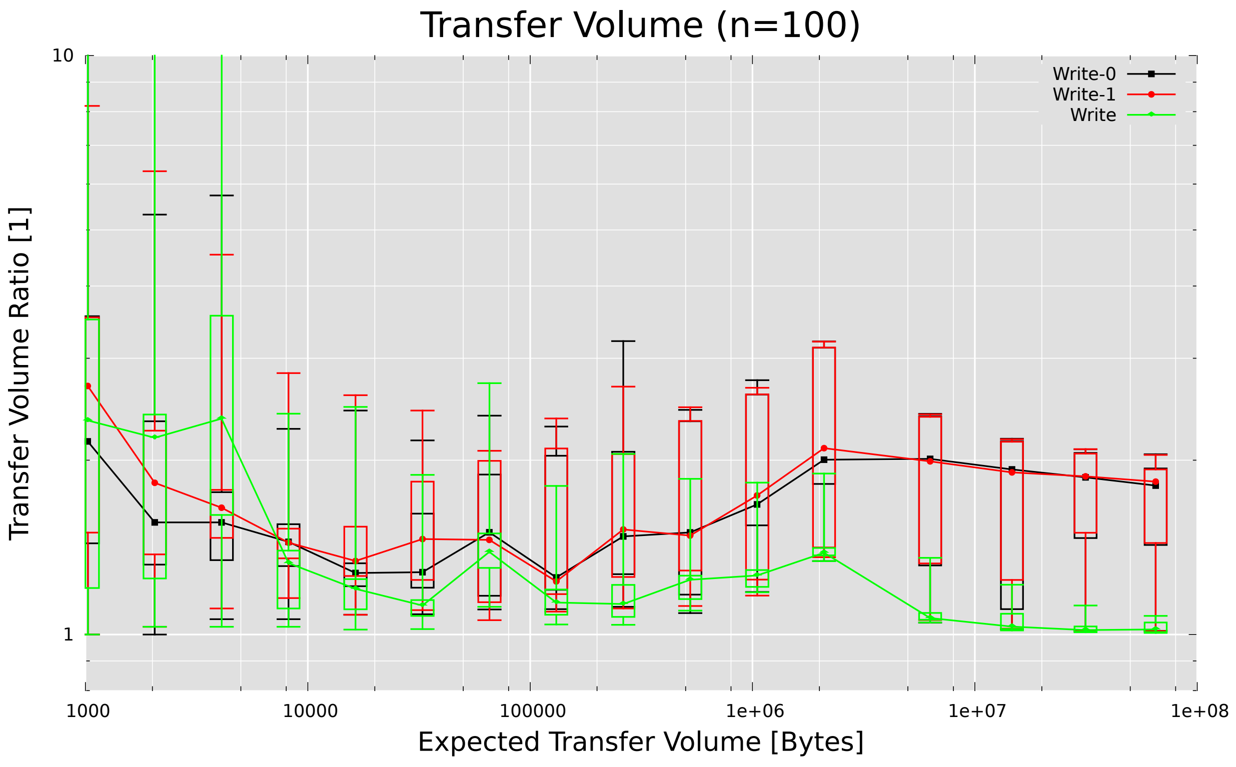


(a) MemBus

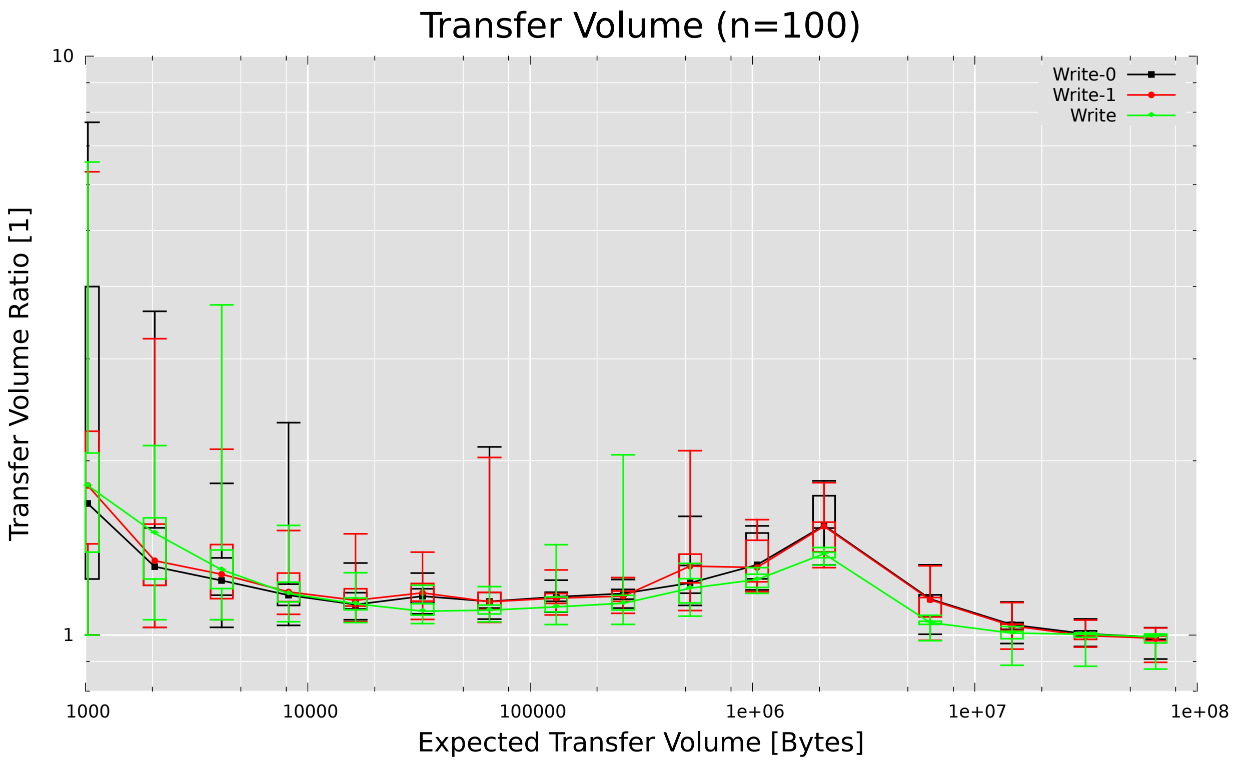


(b) MemL2

Figure 4.1: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Read Kernels running in Parallel (using one Buffer per Kernel)

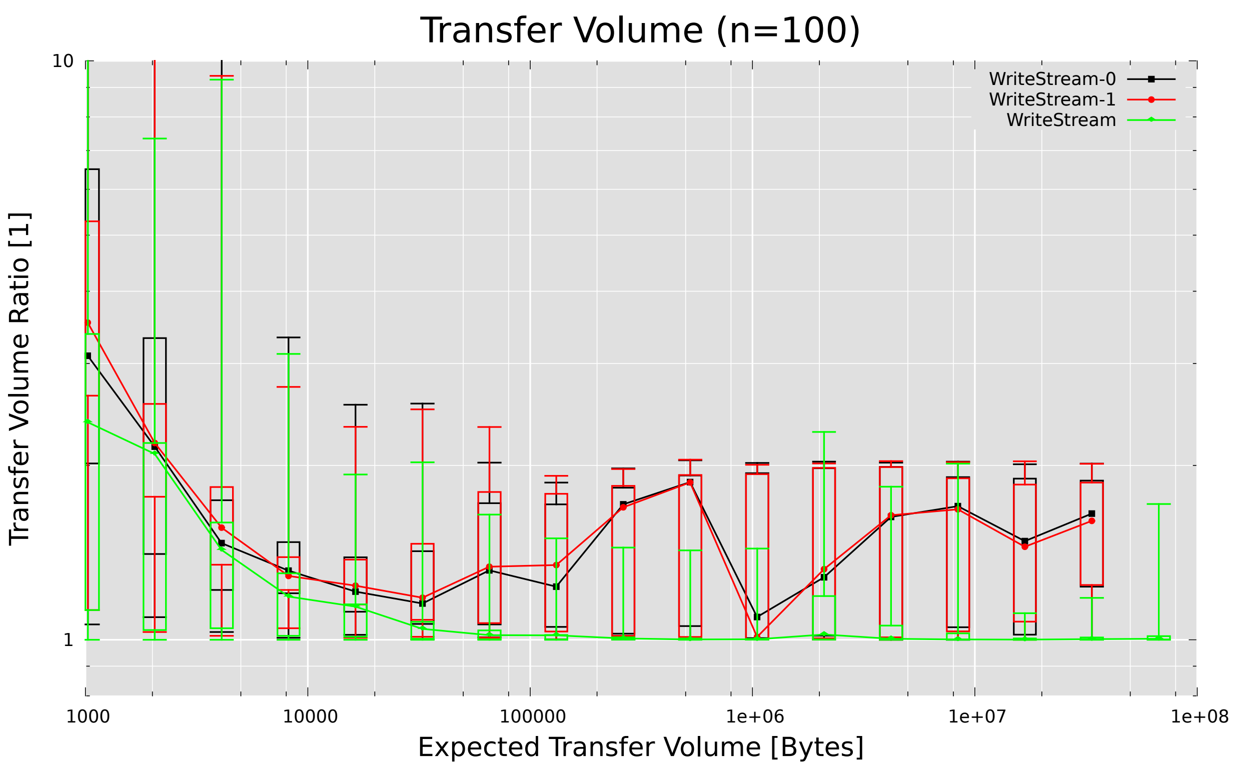


(a) MemBus

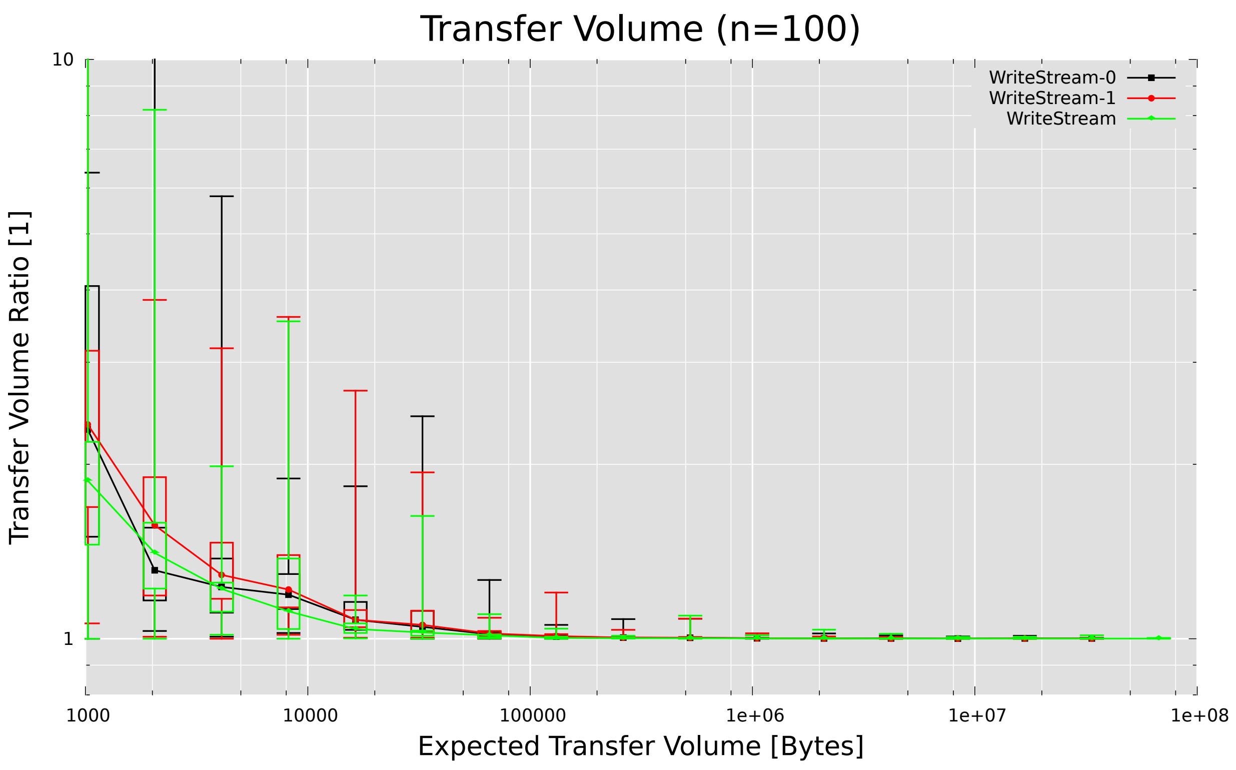


(b) MemL2

Figure 4.2: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Write Kernels running in Parallel (using one Buffer per Kernel)

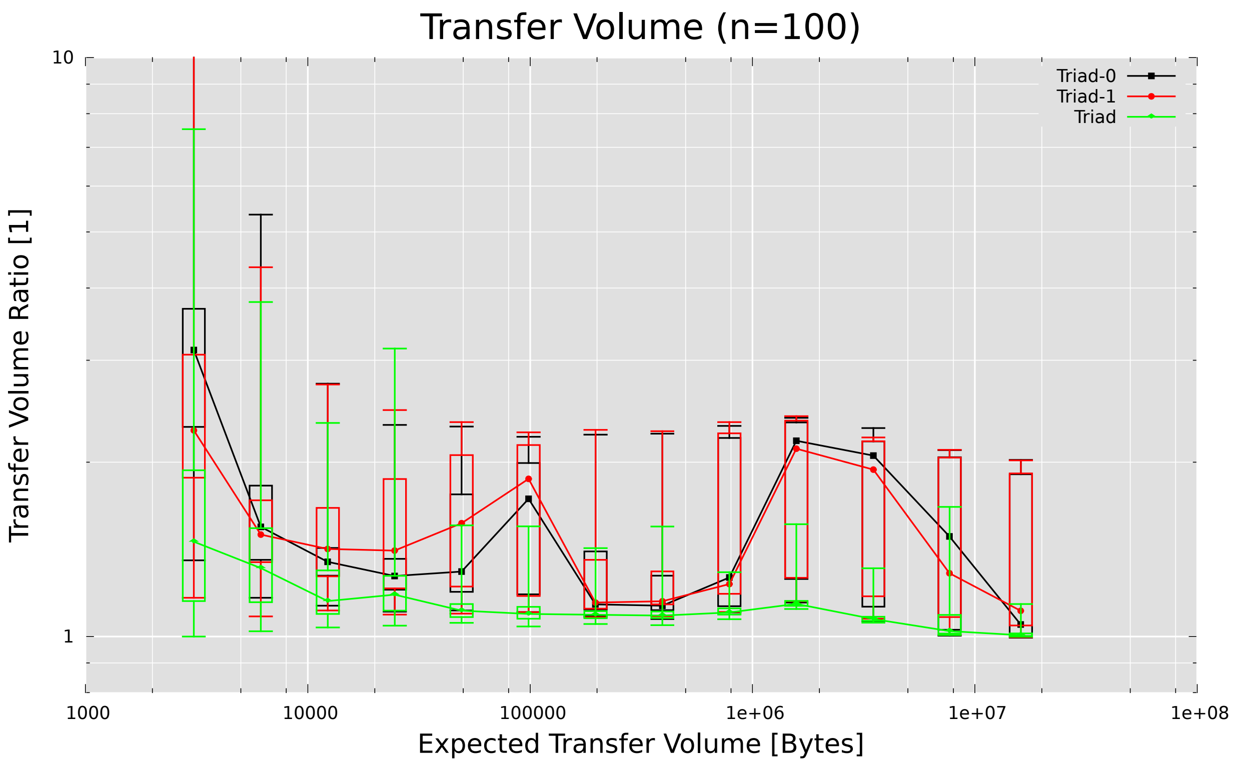


(a) MemBus

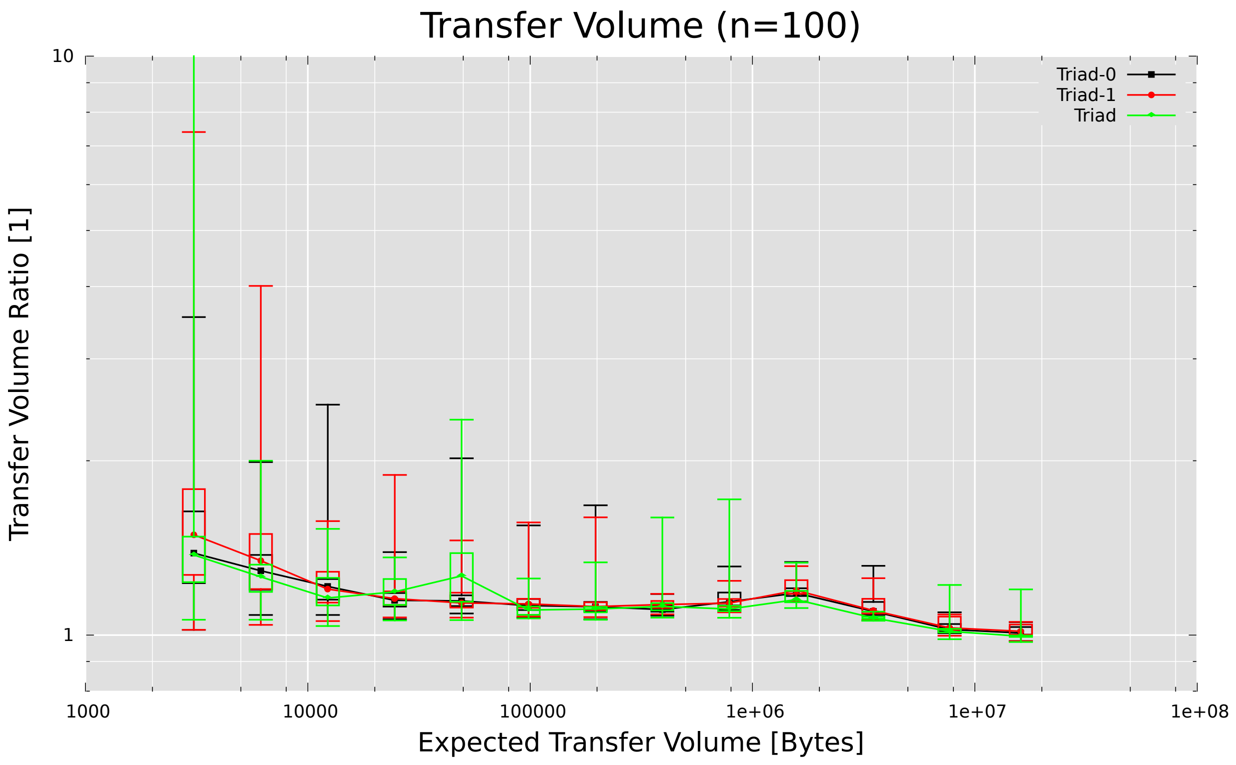


(b) MemL2

Figure 4.3: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Write Kernels using Streaming Stores running in Parallel (using one Buffer per Kernel)



(a) MemBus



(b) MemL2

Figure 4.4: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Triad Kernels running in Parallel (using one Buffer per Kernel)

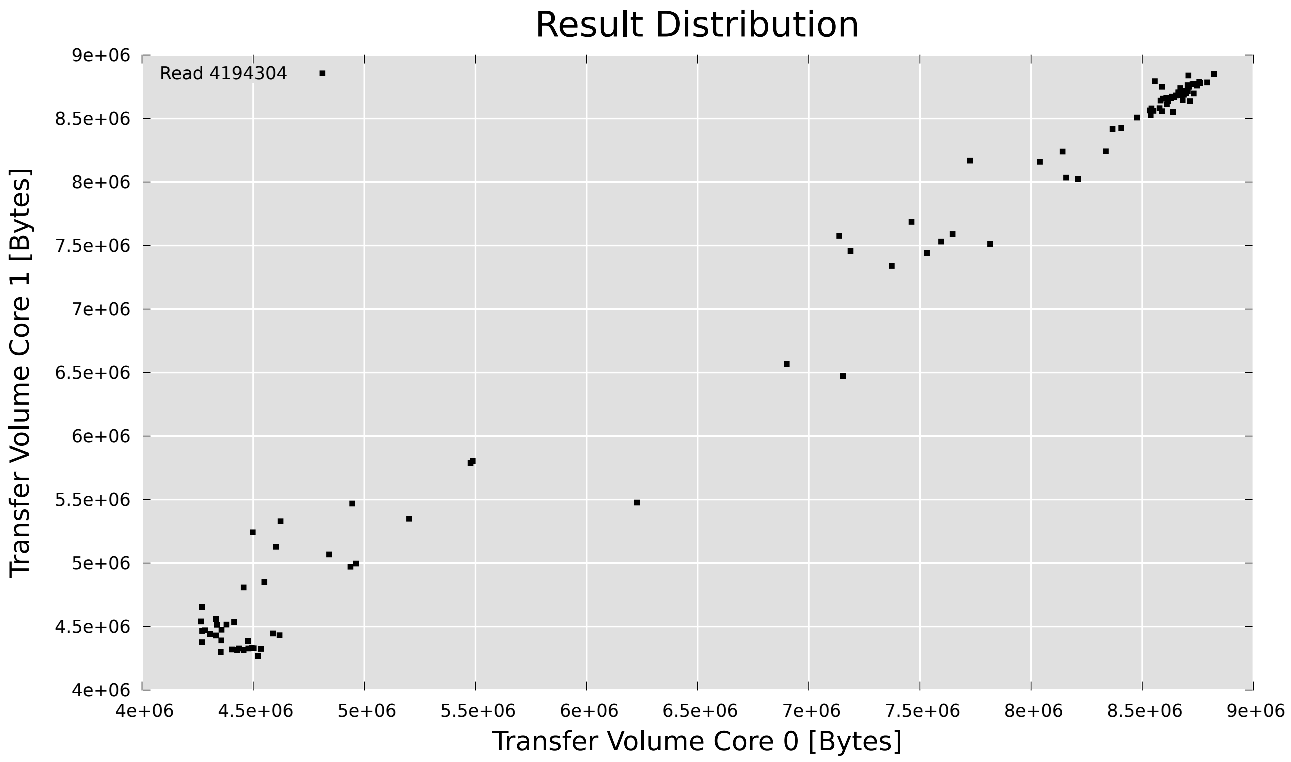


Figure 4.5: Distribution of the Results of the Read Kernel using MemBus

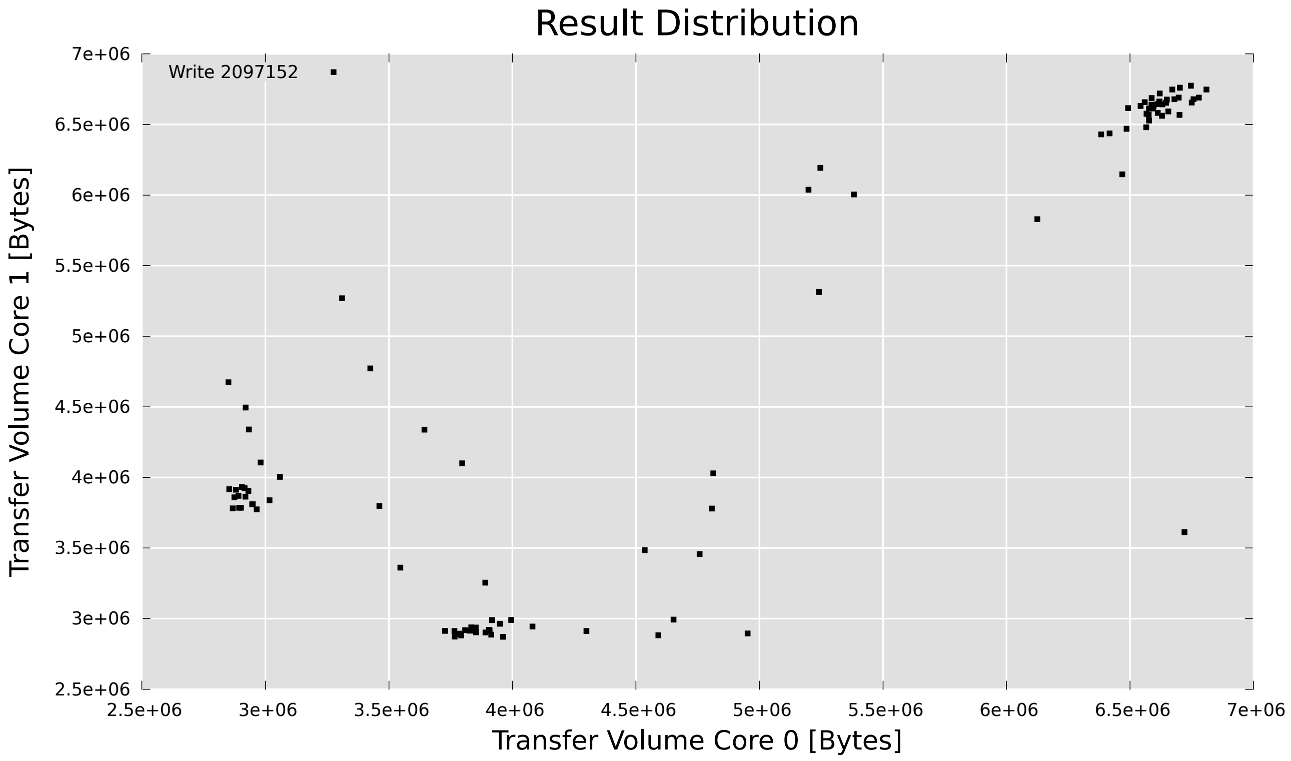


Figure 4.6: Distribution of the Results of the Write Kernel using MemBus

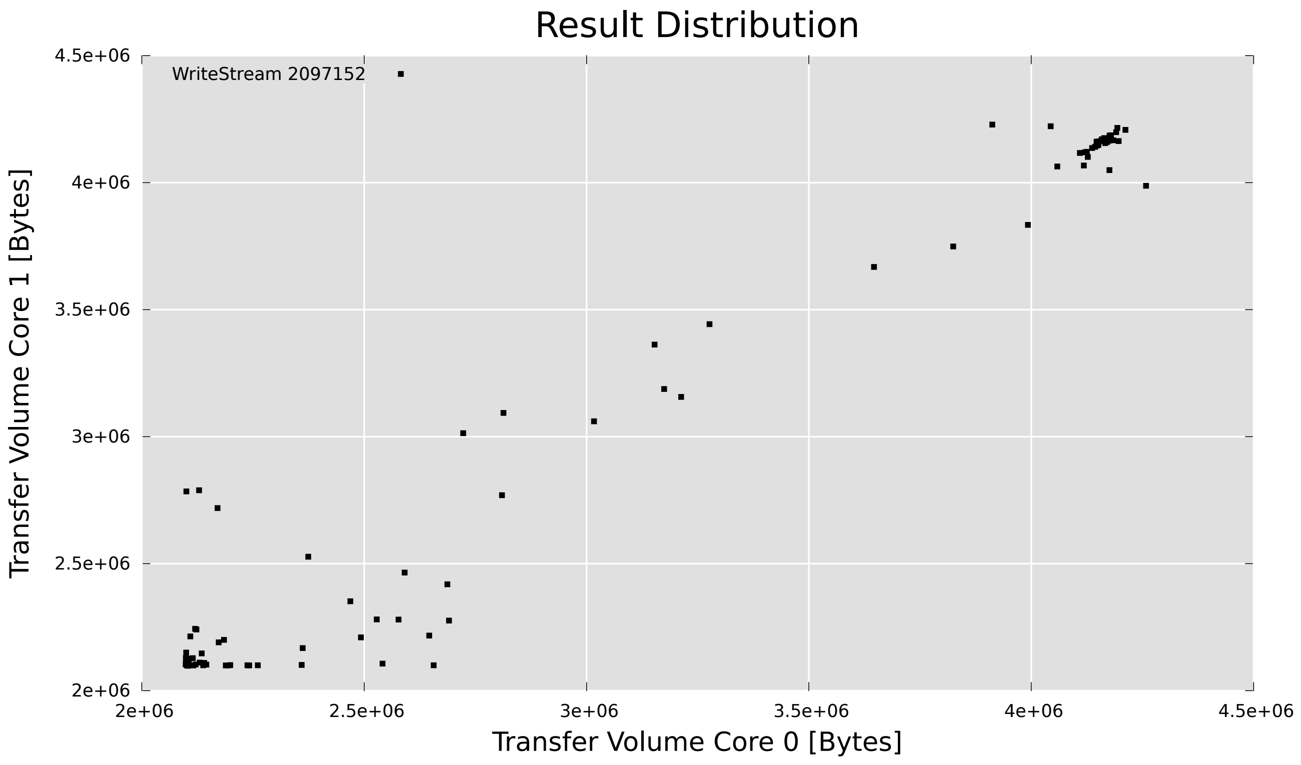


Figure 4.7: Distribution of the Results of the Write Kernel using Streaming Stores measured with the MemBus measurement variant

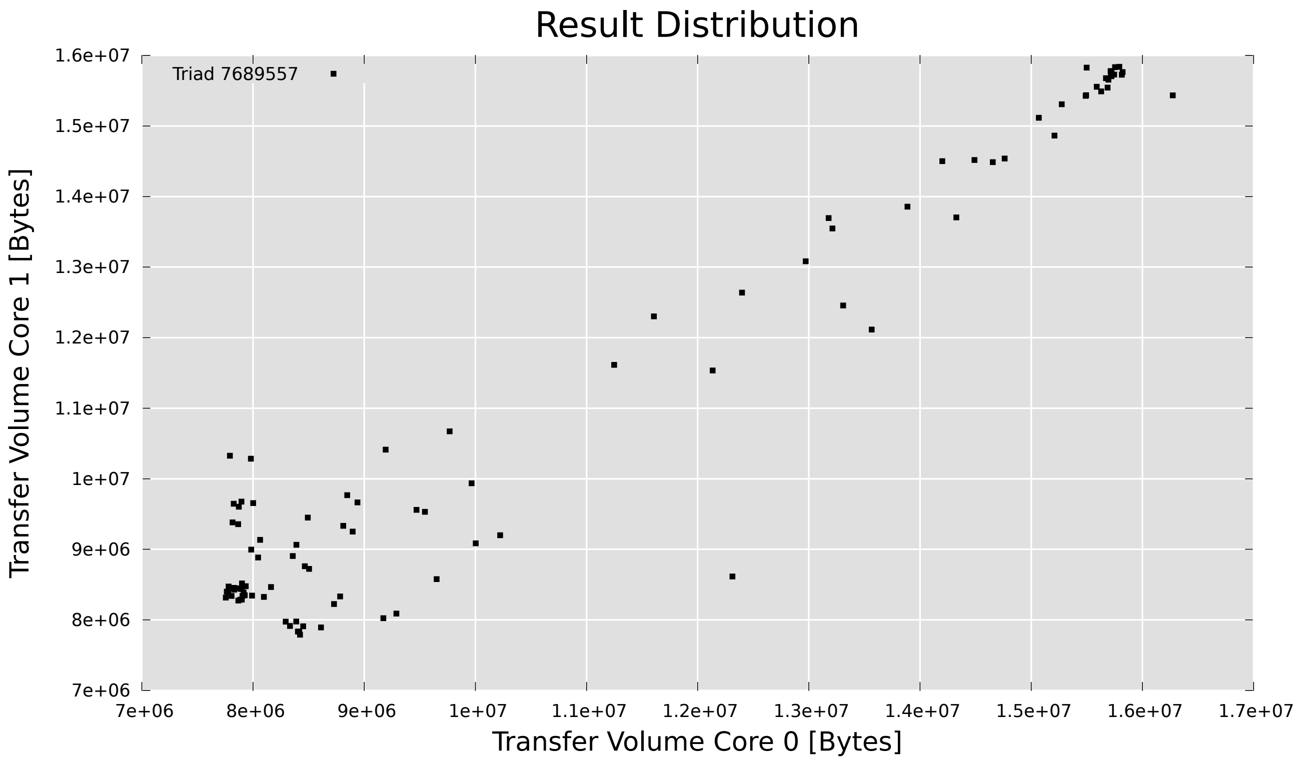


Figure 4.8: Distribution of the Results of the Triad Kernel using MemBus

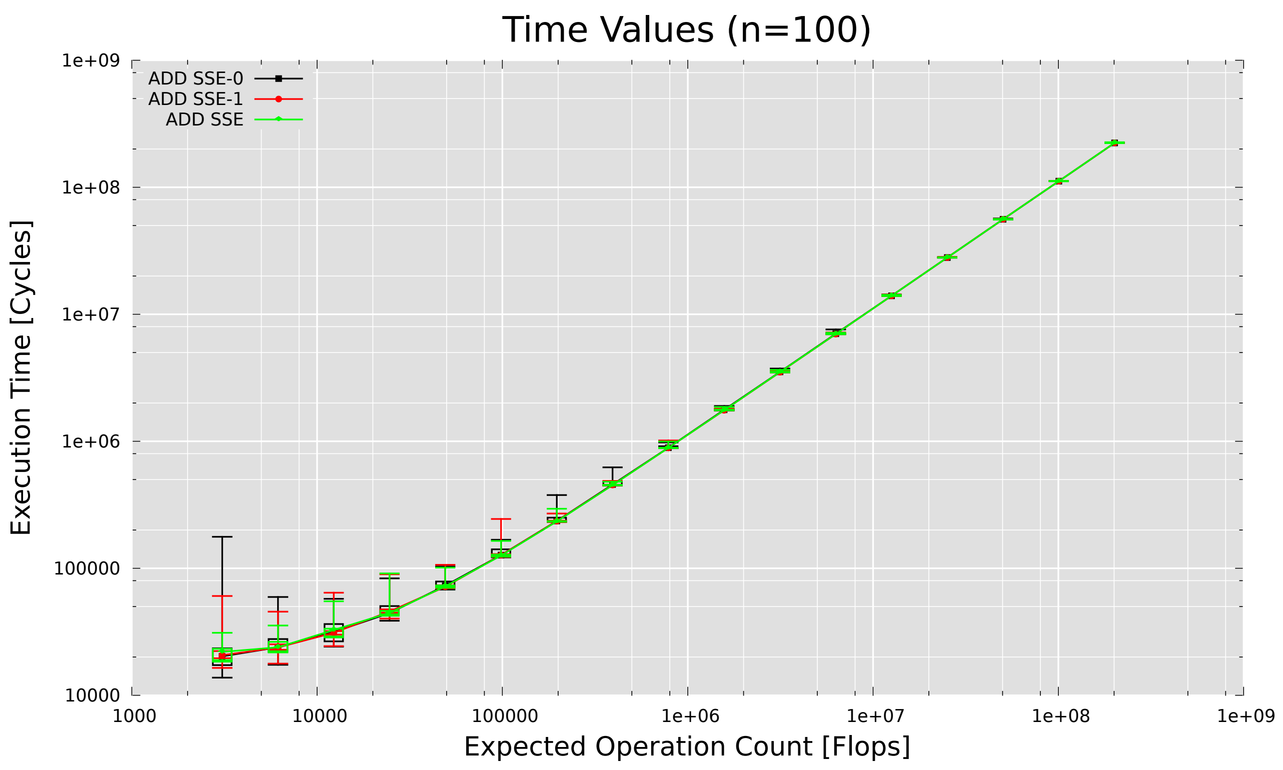


Figure 4.9: Execution Time of two ADD Kernels running at the same time

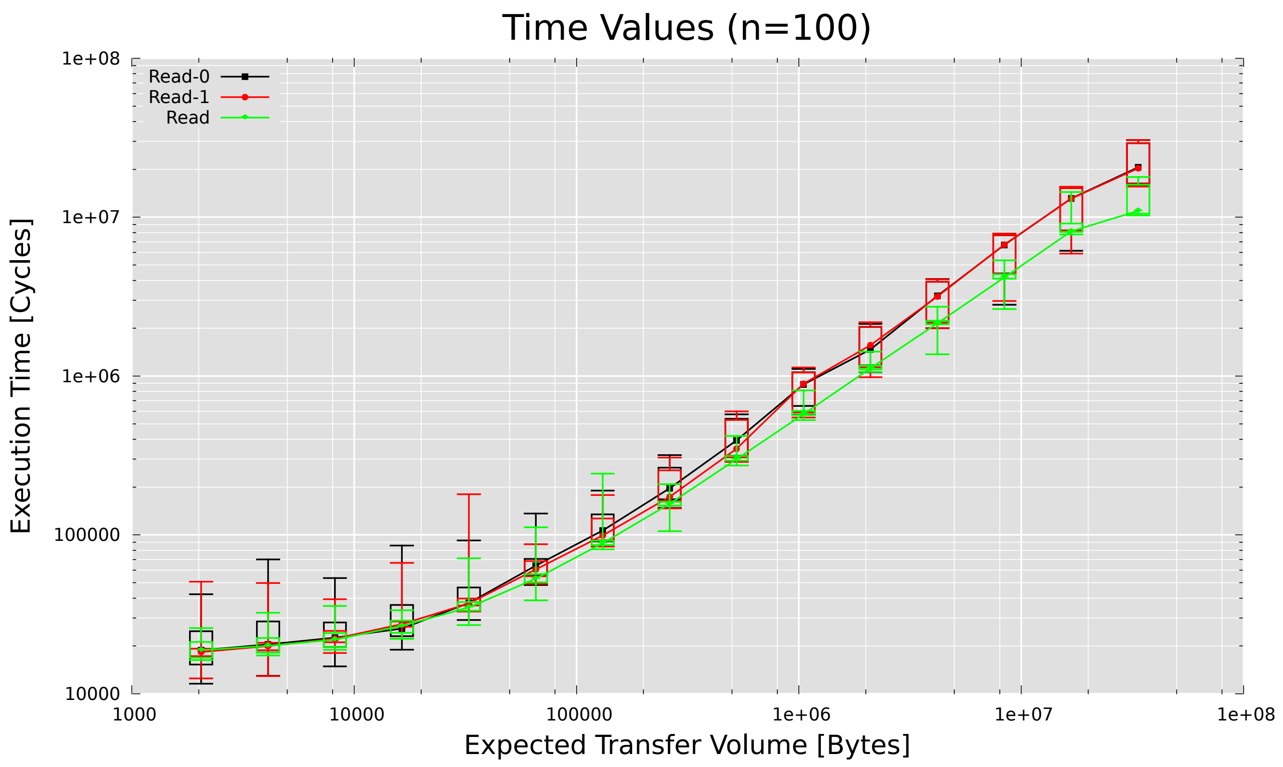


Figure 4.10: Execution Time of two Read Kernels running at the same time

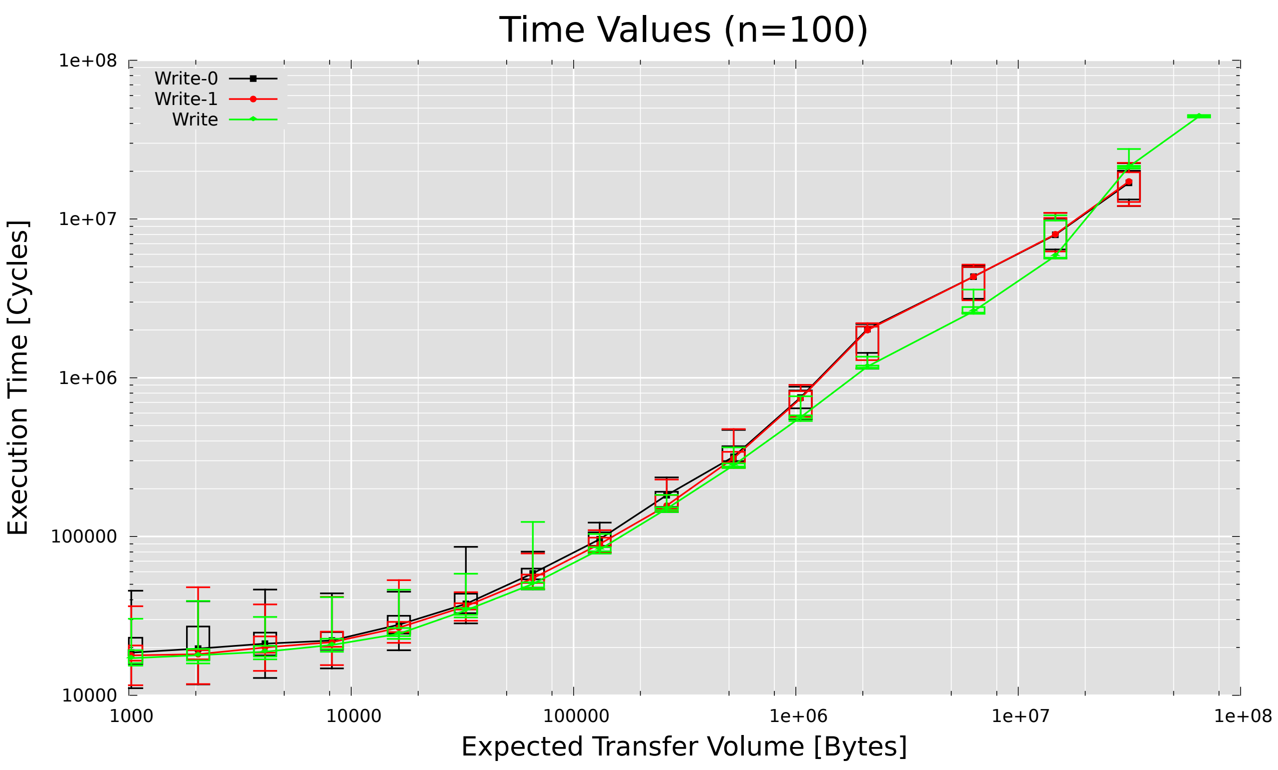


Figure 4.11: Execution Time of two Write Kernels running at the same time

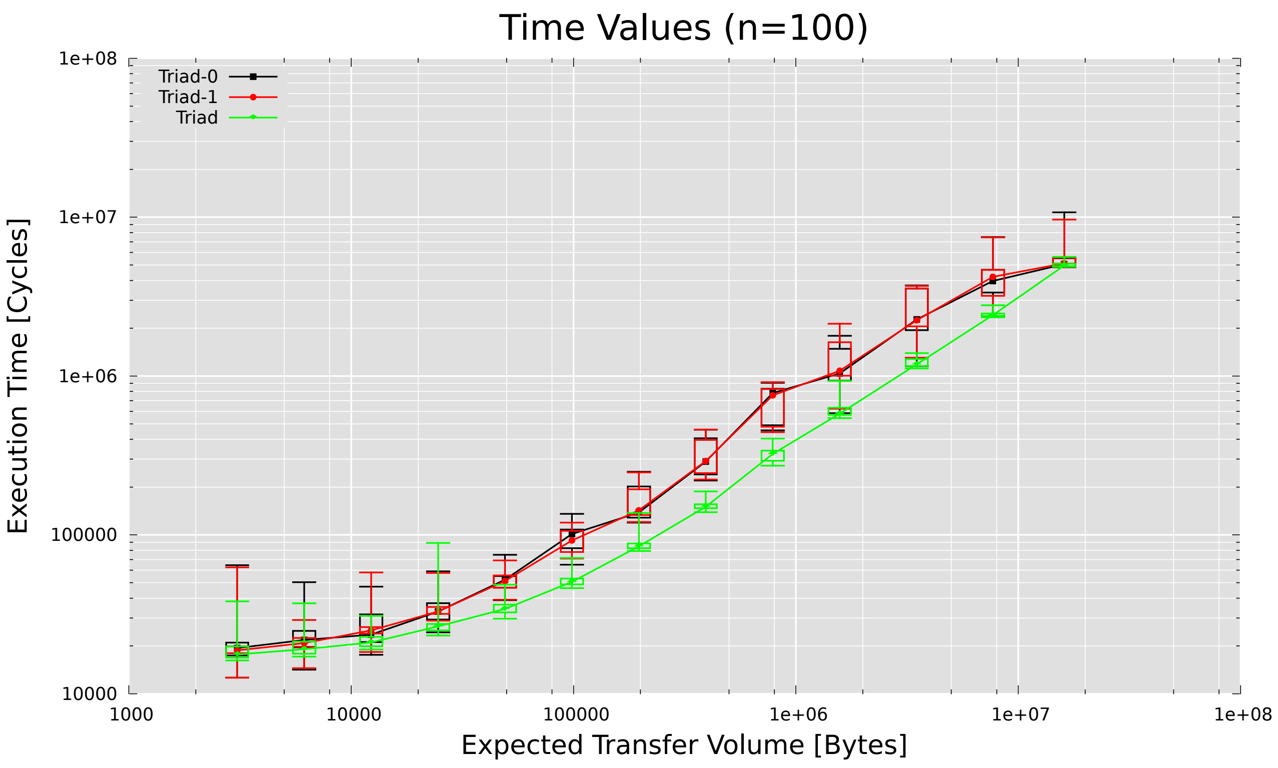


Figure 4.12: Execution Time of two Triad Kernels running at the same time

4.3 Operation Count

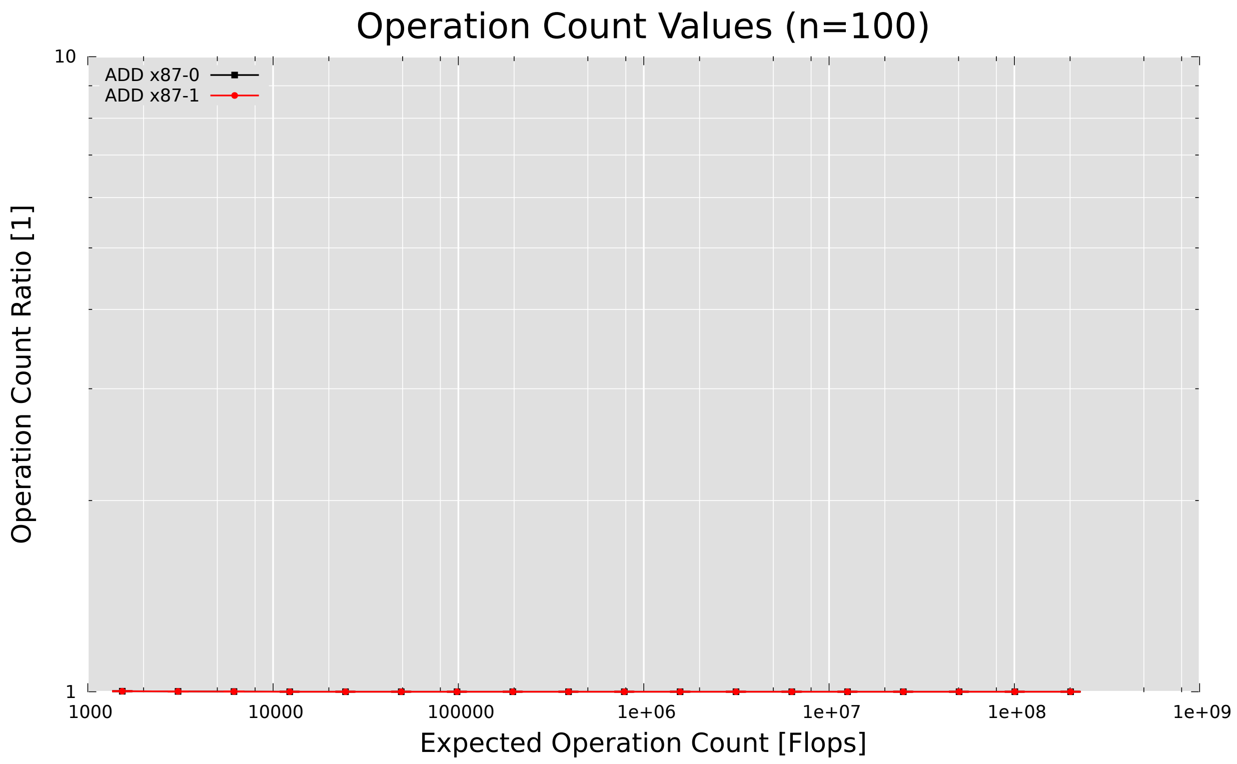


Figure 4.13: Ratio of the Actual Operation Count to the Expected Operation Count for two ADD Kernels running at the same time

As for the single threaded case, measuring the operation count does not pose any problems. (fig. 4.13)

4.4 Handling the Variance

We again use the minimum of $K = 10$ scheme. (see 3.4)

For measuring the transfer volume the precision of the MemL2 variant is superior to the precision of the MemBus variant. Using the MemL2 variant, we get a variation below 10% if the expected transfer volume is above 10KB.

For the Read, Write and Triad kernel the overhead is between 10% and 20% for buffers below 10MB and goes towards zero for transfer volumes exceeding 10MB. (figs 4.14, 4.15 and 4.17).

The streaming writes can be measured with high precision and accuracy for transfervolumes beyond 100KB using the MemL2 variant. The MemBus variant has a high variance for small and large buffers. Between 10KB and 10MB the results fall within 10% of the expected values (fig. 4.16).

When measuring memory intensive kernels we measure huge variations of the execution time for small buffer sizes. If the expected transfer volume exceeds 20KB, the variations drop to around 10% and become large again for transfer volumes exceeding 10MB. (figs. 4.18, 4.19, 4.20 and 4.21)

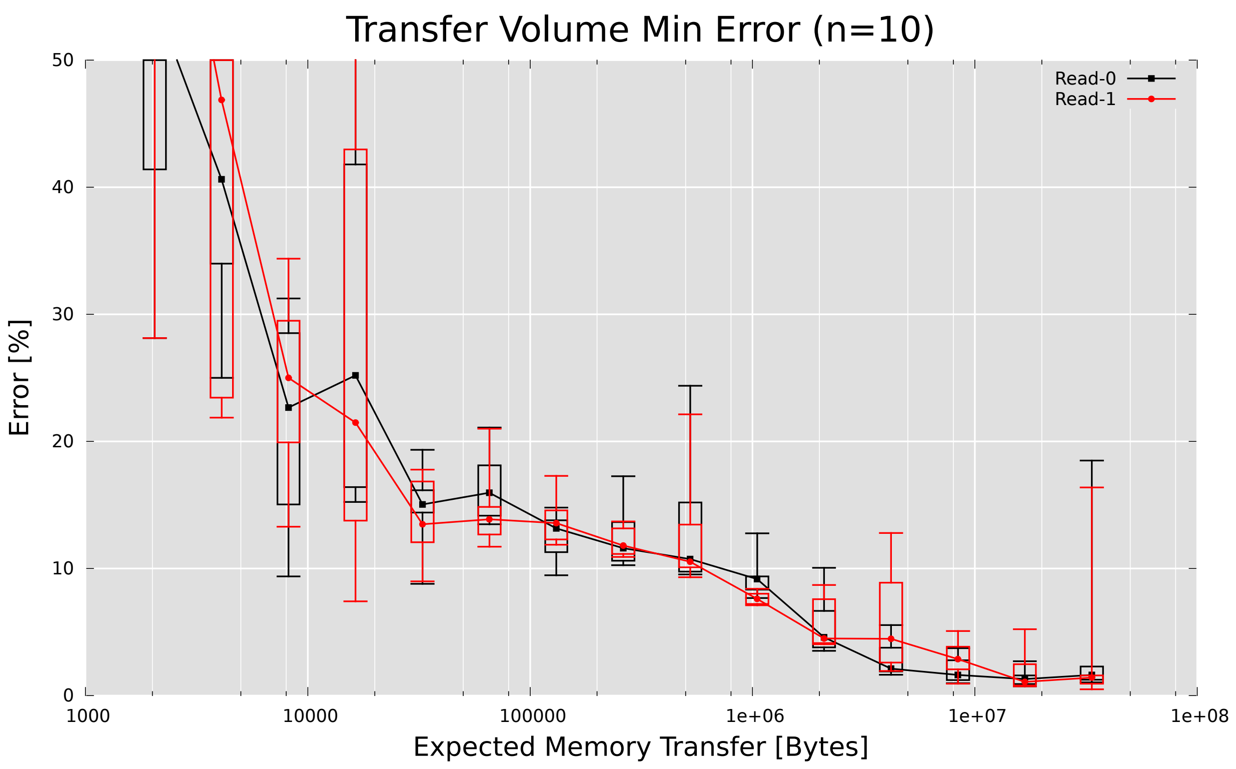
The variations observed when running two ADD kernels in parallel are small except for iteration counts below 100'000. (fig. 4.22)

As expected, the results for the operation count are perfect (fig. 4.23)

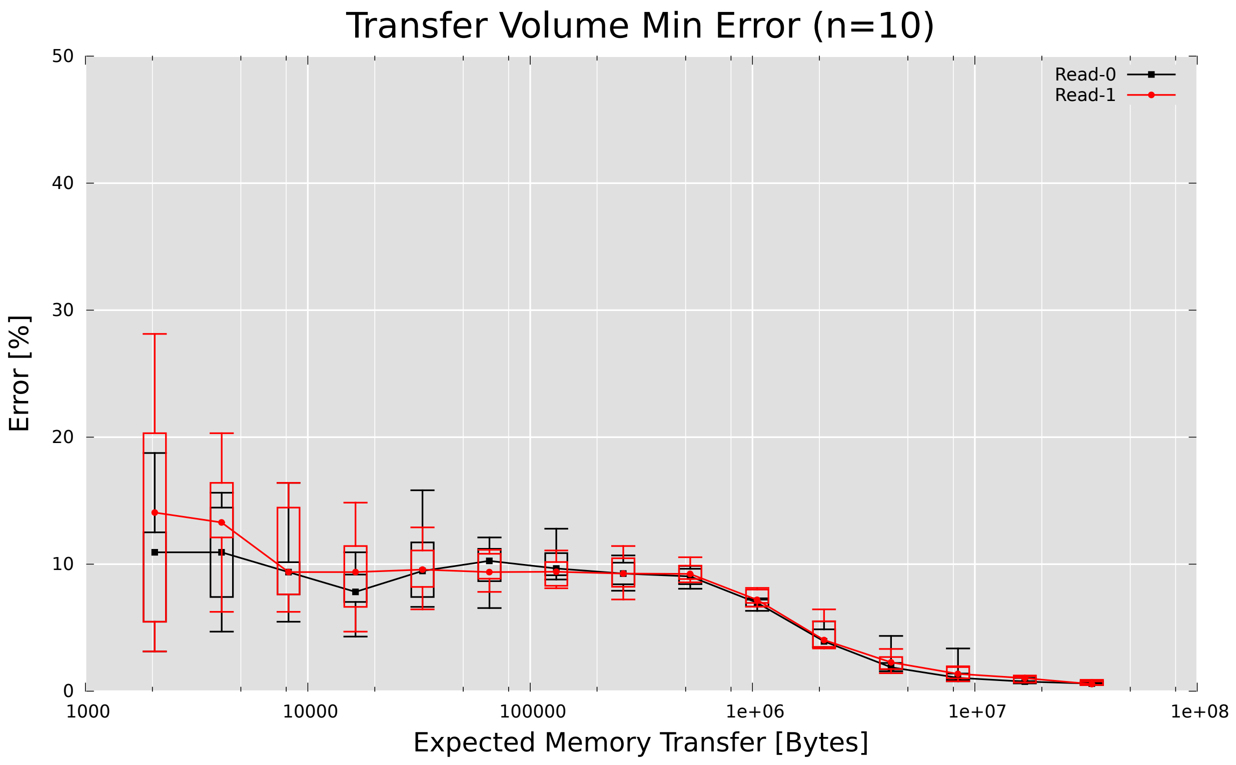
Summarizing our results, transfer volumes can be measured with a precision of 10%. They are overestimated by about 20% for transfer volumes below 10MB.

The precision of the measured execution times depends heavily on the type of kernel measured. Memory intensive kernels are measured with a precision of about 10% for transfer volumes between 10KB and 10MB and very high variances outside of that range. Arithmetic kernels can be measured with a high precision.

The operation count measurements are very accurate and precise.

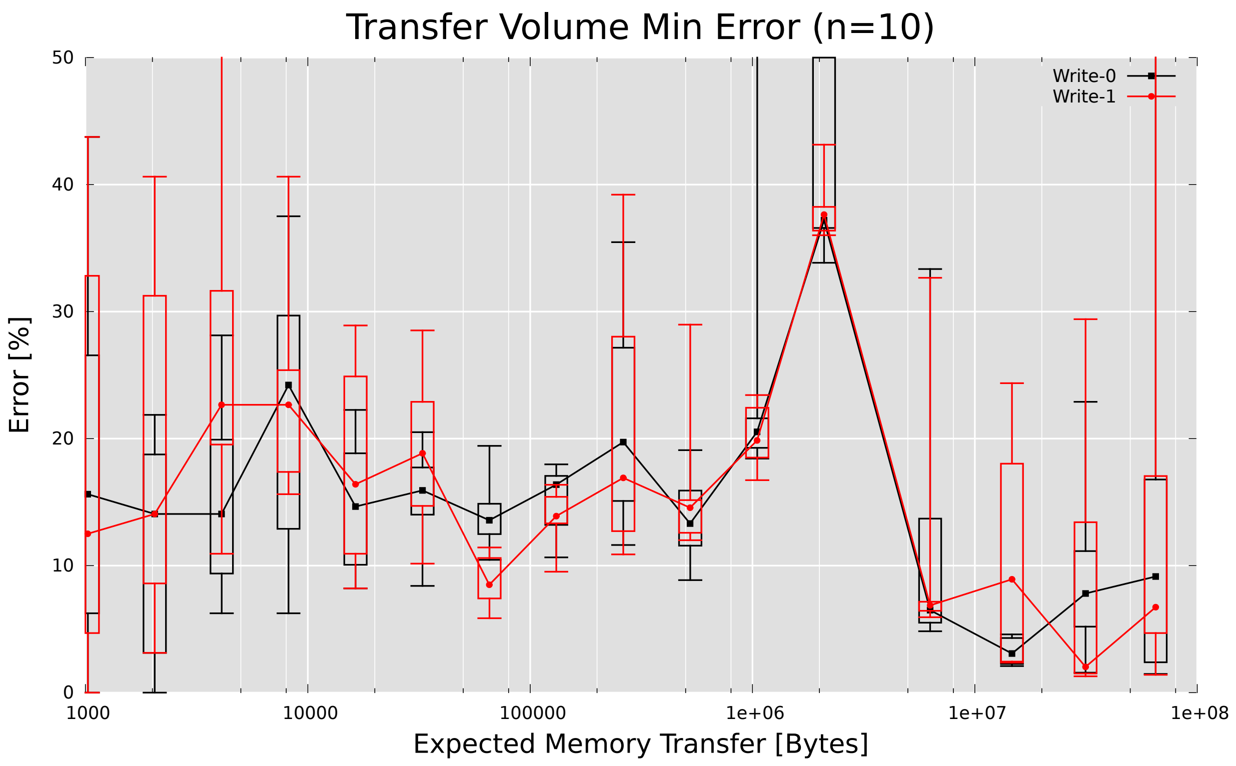


(a) MemBus

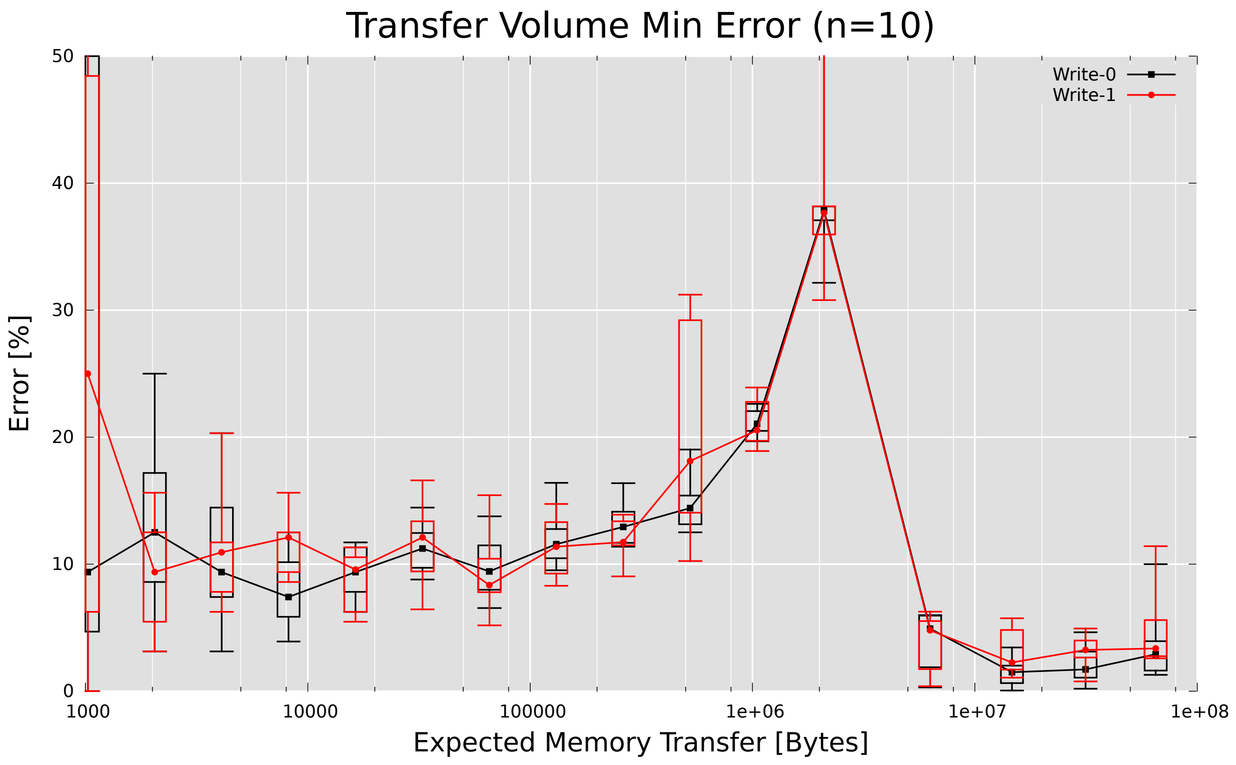


(b) MemL2

Figure 4.14: Error of the Transfer Volume of two Read Kernels running at the same time with $K = 10$: err $\left(\frac{\tau_{10}}{\Theta}\right)$

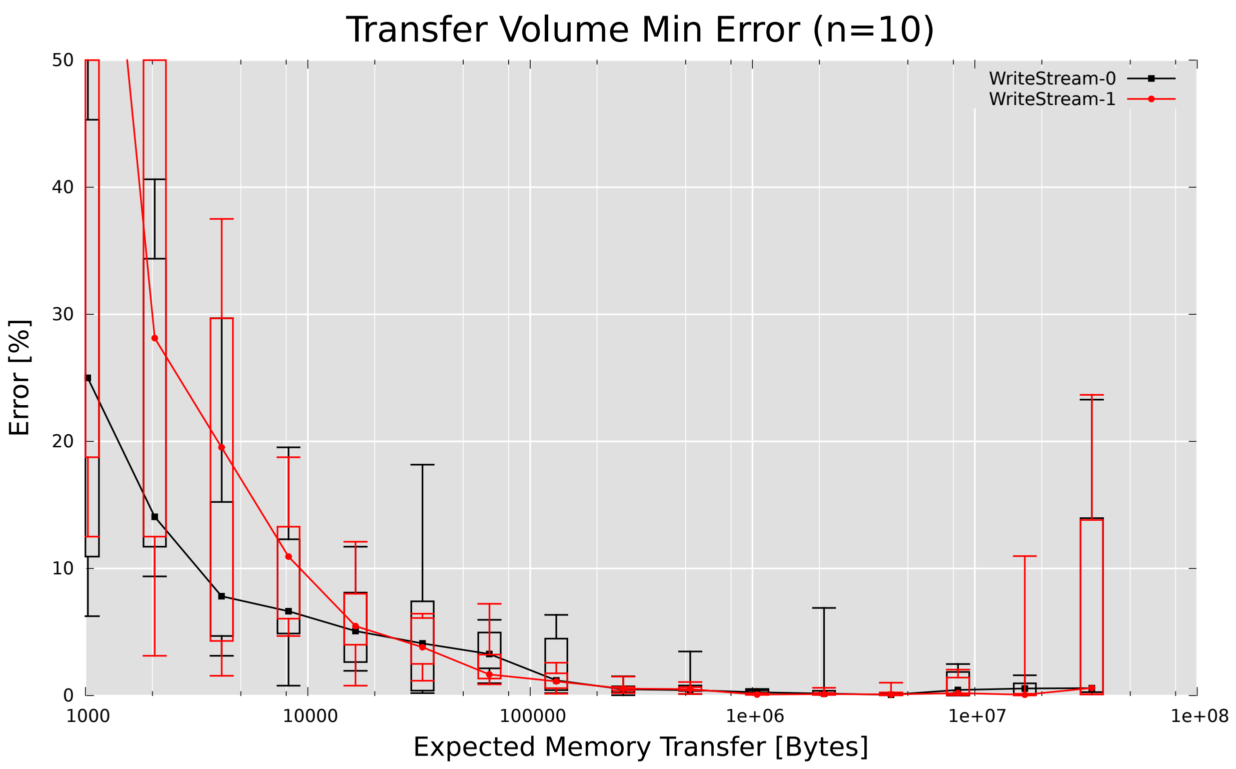


(a) MemBus

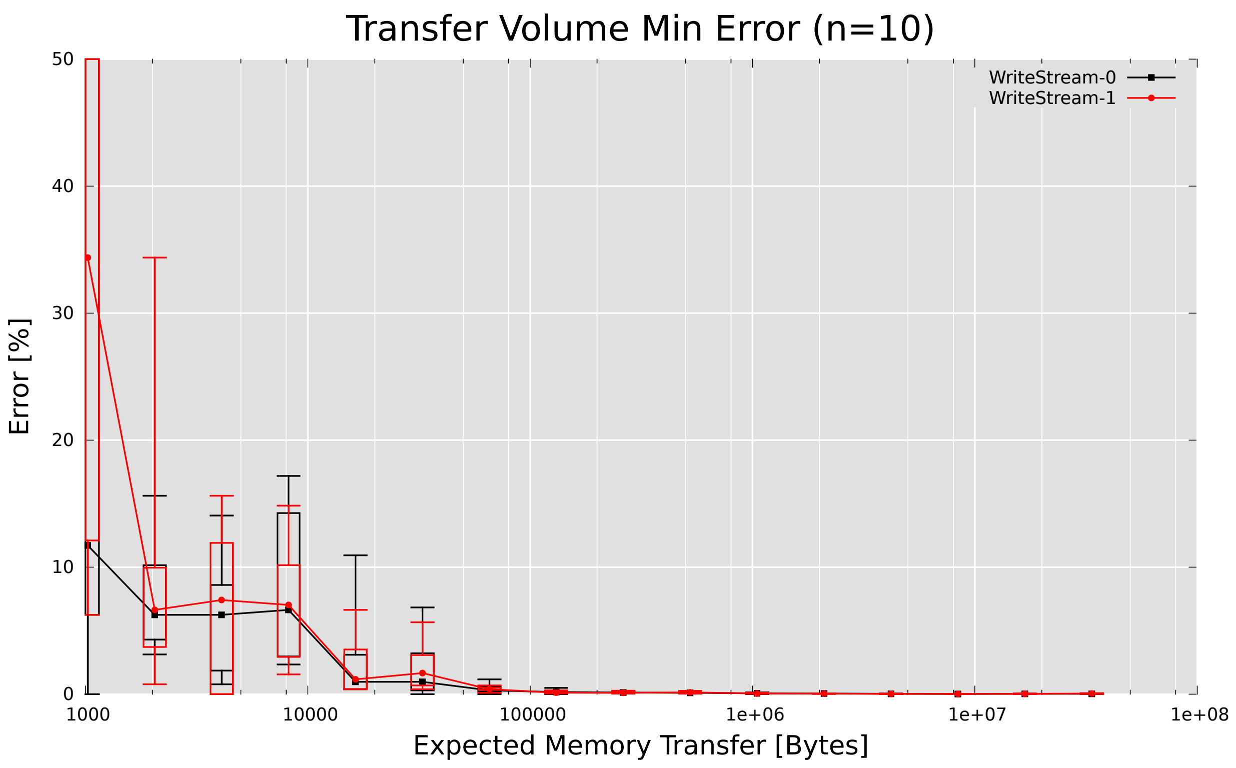


(b) MemL2

Figure 4.15: Error of the Transfer Volume of two Write Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\tau_{10}}{\Theta}\right)$

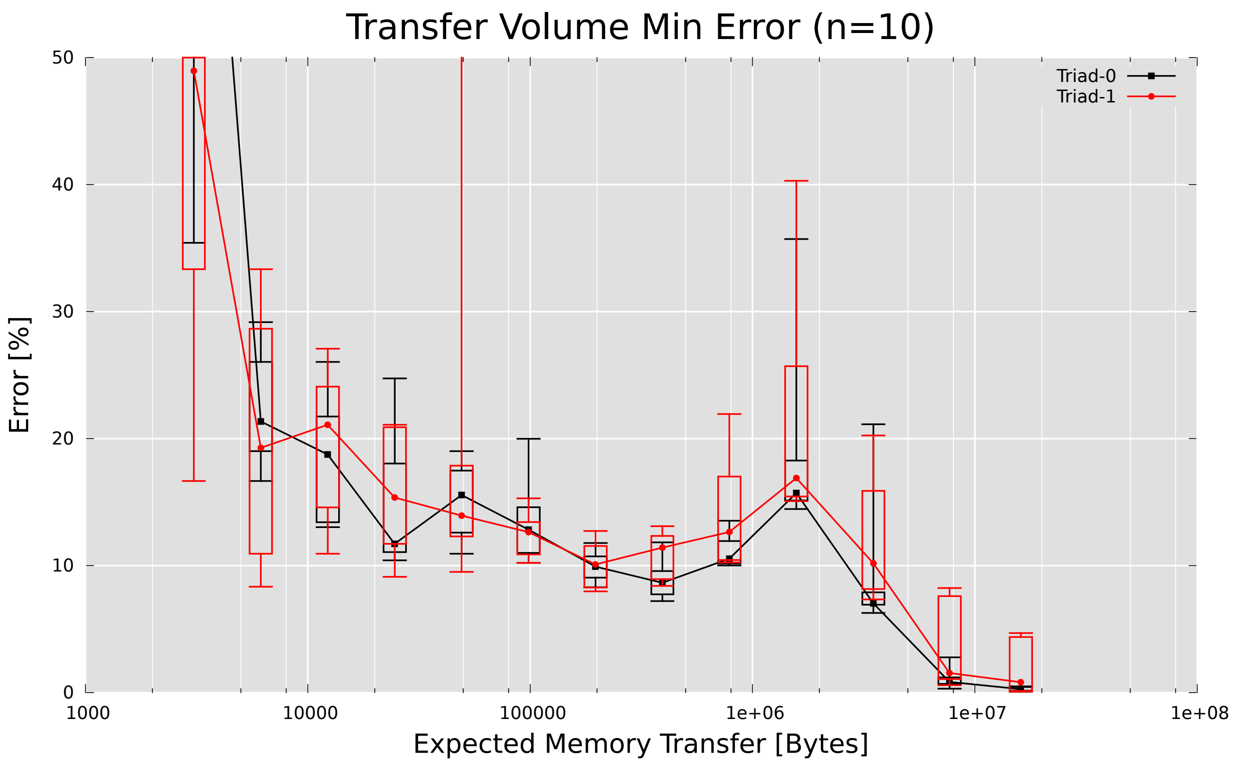


(a) MemBus

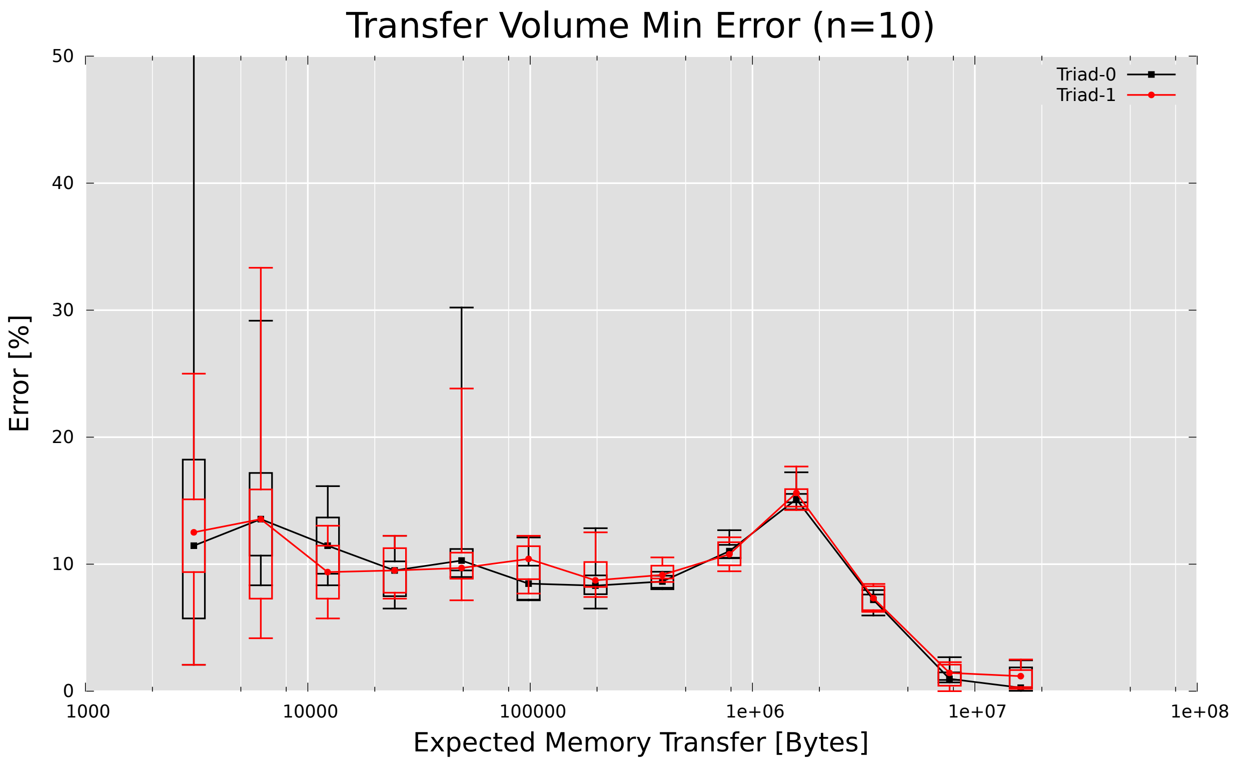


(b) MemL2

Figure 4.16: Error of the Transfer Volume of two Write Kernels using Streaming Stores running at the same time with $K = 10$: $\mathrm{err}\left(\frac{\tau_{10}}{\Theta}\right)$



(a) MemBus



(b) MemL2

Figure 4.17: Error of the Transfer Volume of two Triad Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\tau_{10}}{\Theta}\right)$

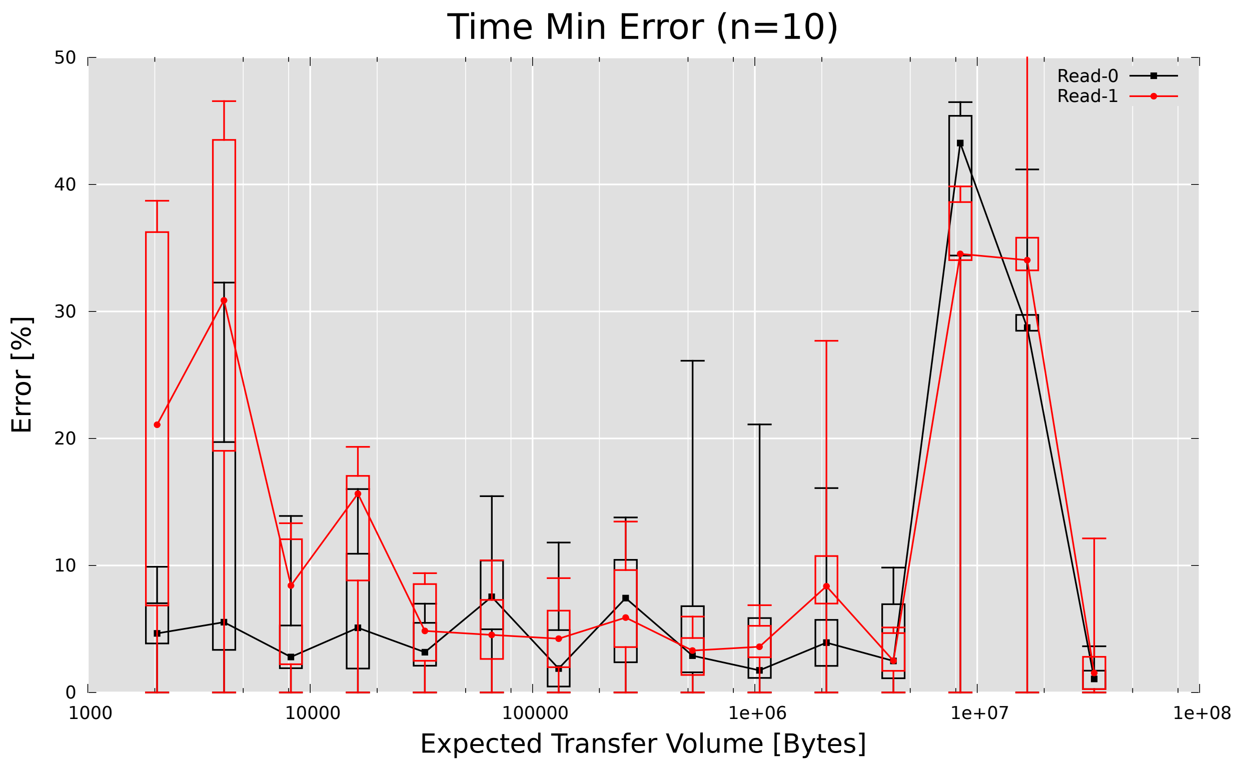


Figure 4.18: Error of the Execution Time of two Read Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

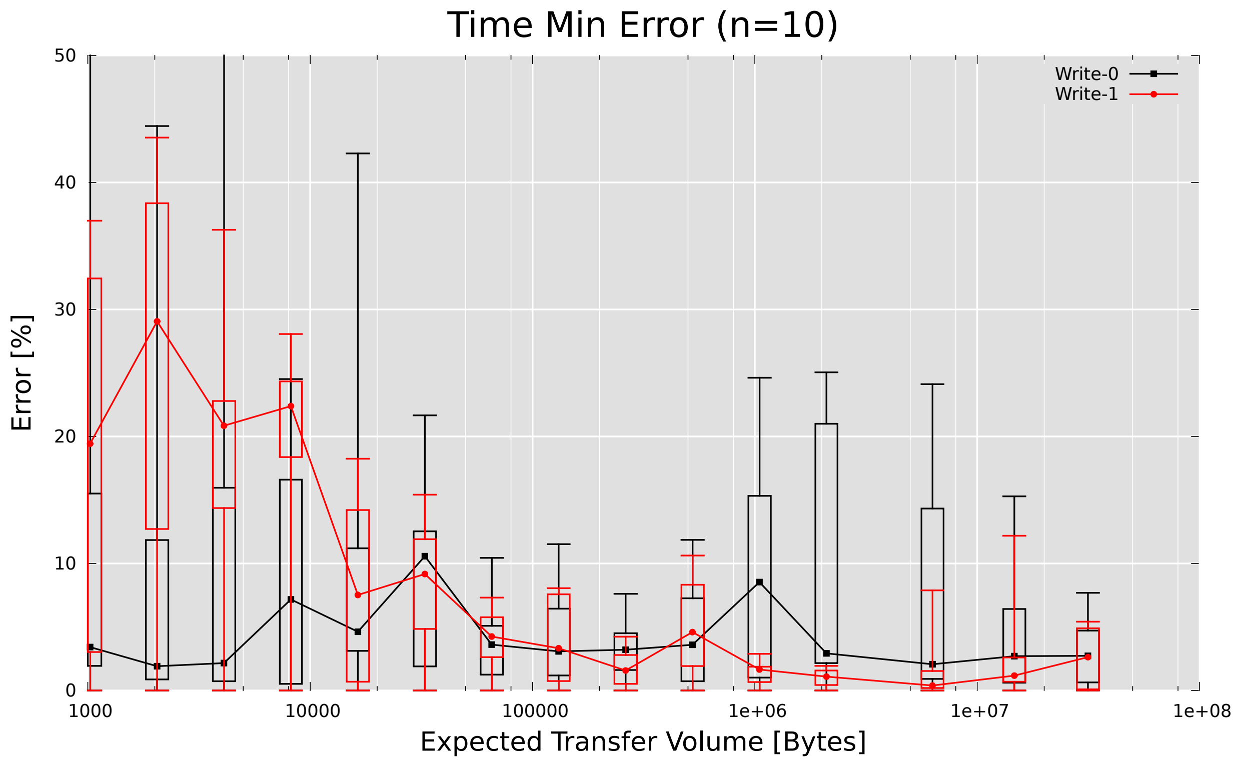


Figure 4.19: Error of the Execution Time of two Write Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

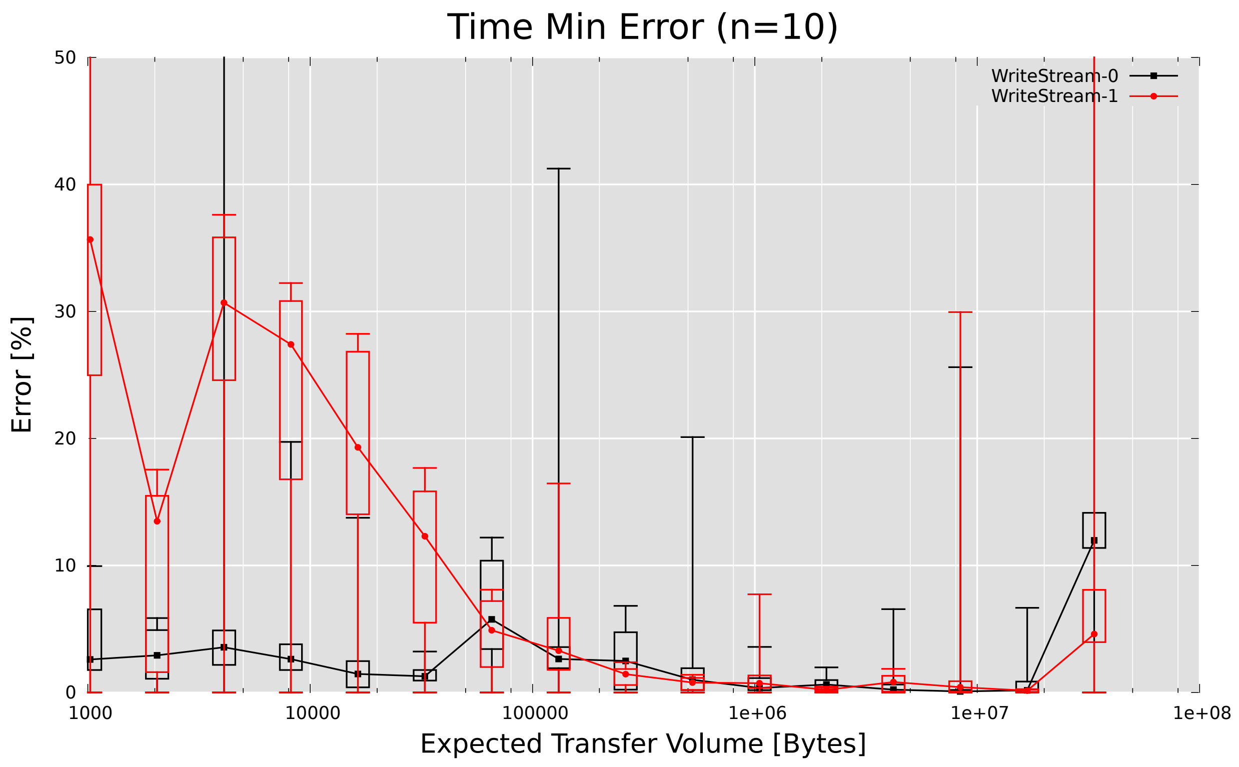


Figure 4.20: Error of the Execution Time of two Write Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

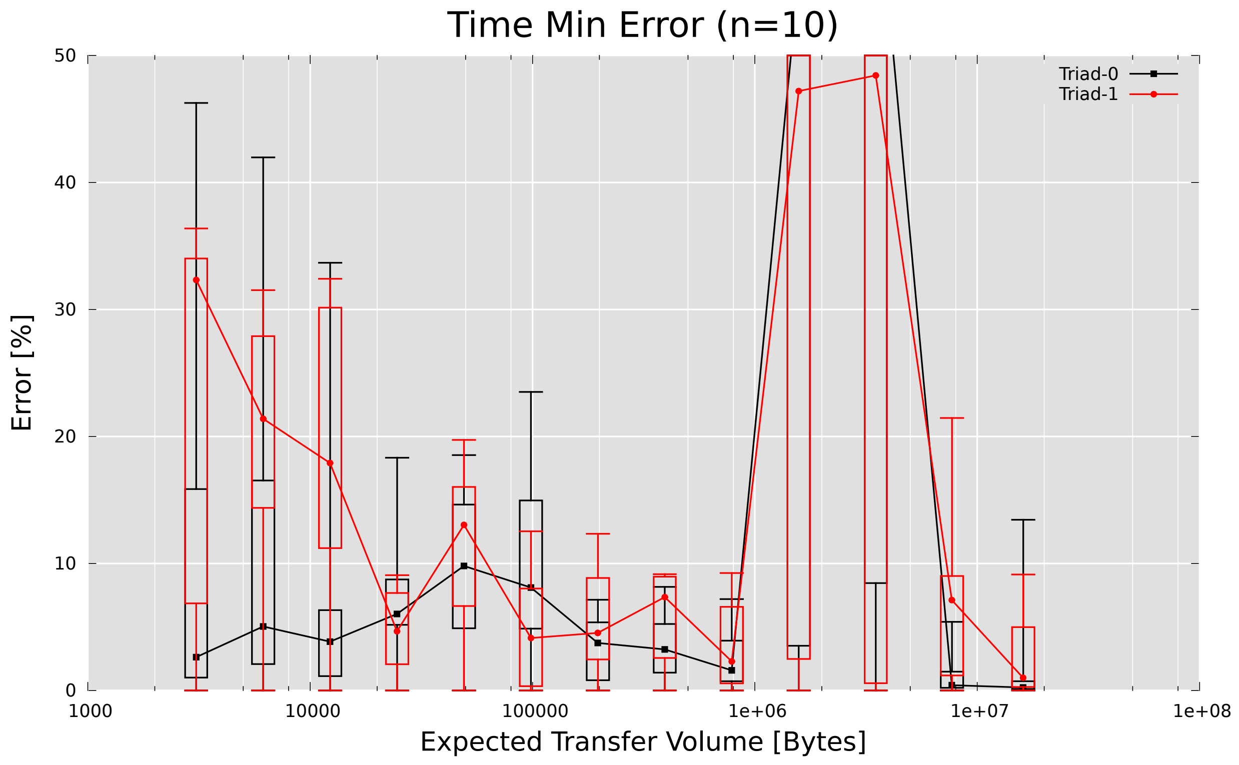


Figure 4.21: Error of the Execution Time of two Triad Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

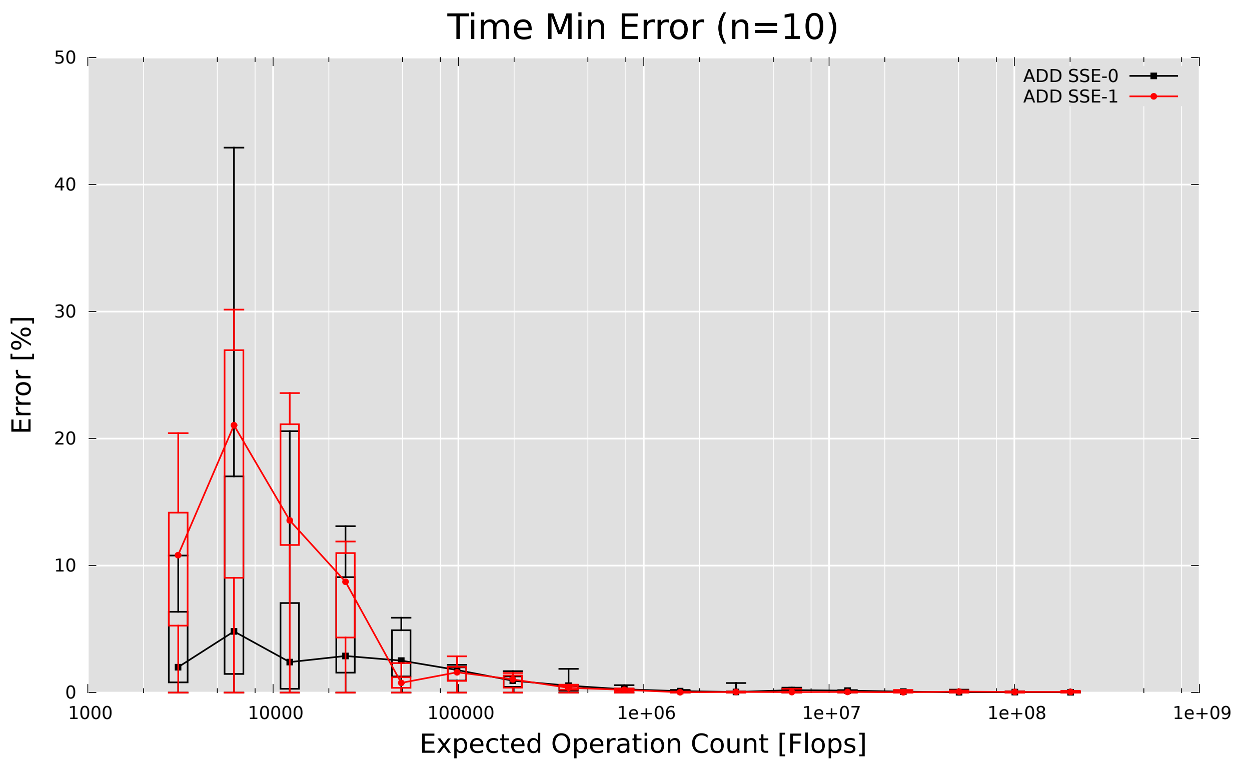


Figure 4.22: Error of the Execution Time of two ADD Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

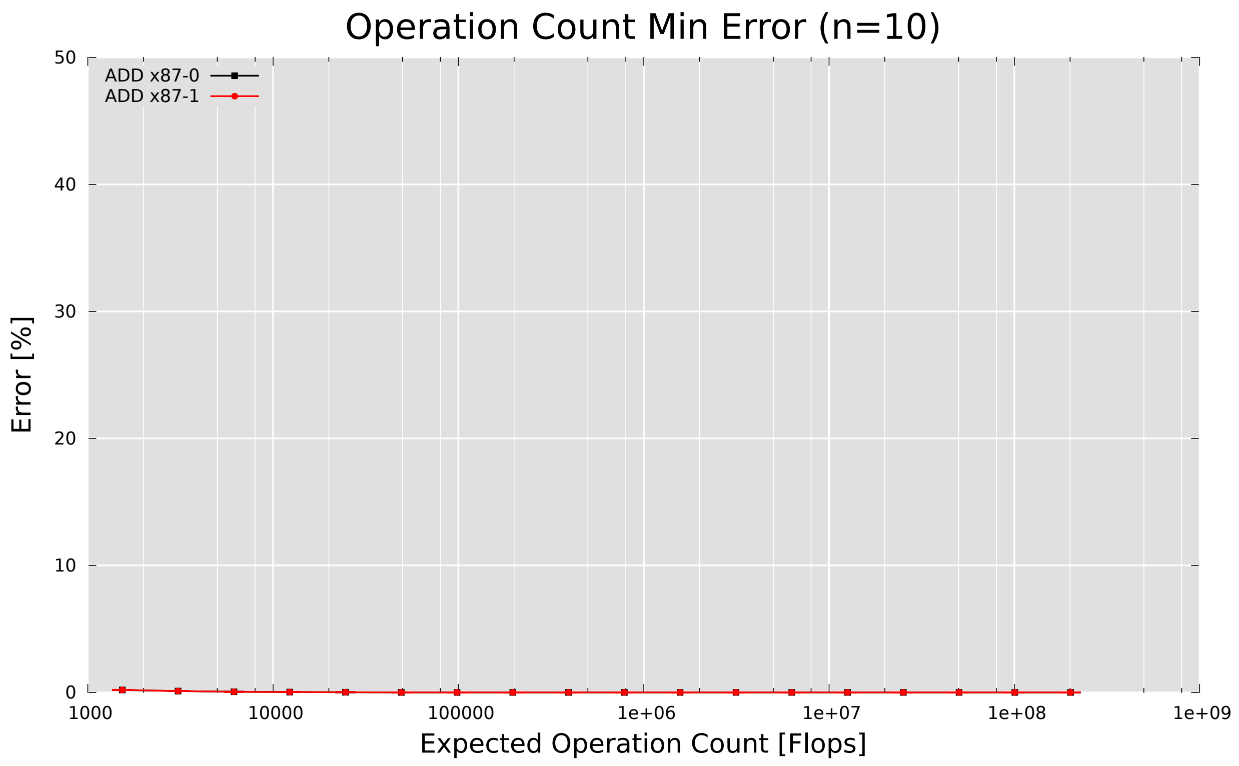


Figure 4.23: Error of the Operation Count of two ADD Kernels running at the same time with $K = 10$: $\text{err}(\frac{\text{Actual Operation Count}_{10}}{\text{Expected Operation Count}})$

Chapter 5

Finding Performance and Bandwidth Ceilings

Roofline plots show ceilings for the performance and the bandwith of the measurement system. The values for the ceilings can be measured or derived from the system specification. We will use both approaches and compare the results.

By the manual [12] CPUs using the Yonah architecture can issue one double precision addition every cycle and one multiplication every two cycles. The latencies are given as 3 cycles for the addition and 5 cycles for the multiplication.

The latency specifies how many cycles after issuing an operation the result becomes available. If the result of a first operation used as input for a second operation, the second operation has to wait for the first operation to complete.

There is no difference between the x87 and the SSE instruction set, since the same execution units are used for both. The performance of SSE code can be slightly better, since more registers are available and fewer instructions need to be decoded.

The theoretical ceiling for pure additions is 1 flop/cycle and 0.5 flop/cycle for pure multiplications.

It should be possible to use the multiplier and the adder at the same time, so if there are two additions per multiplication it should be possible to reach 1.5 flop/cycle.

We implemented a heavily optimized micro benchmark for pure adds, pure multiplications and a mix of additions and multiplications. For the additions we used the following loop:

```
double r[DLP];
...
for (long i = 0; i < iterations; i++) {
    for (int p = 0; p < UNROLL; p++) {
        for (int j = 0; j < DLP; j++) {
            r[j] += 1;
        }
    }
}
```

We allocated an array of DLP doubles. In each iteration, we add 1 to each of the doubles in the array. The use of multiple accumulators avoids the latency and makes it possible to keep multiple instructions in flight. For the SSE implementation we did a straight forward vectorization of the code using SSE intrinsics.

For the multiplication we use a very similar loop. We just replaced the addition with a multiplication.

```
double r[DLP];
...
for (long i = 0; i < iterations; i++) {
    for (int p = 0; p < UNROLL; p++) {
        for (int j = 0; j < DLP; j++) {
            r[j] *= base;
        }
}
}
```

Finally, we created a mix with two additions per multiplication:

```
double mulR[DLP];
double addR[DLP*2];
...
for (long i = 0; i < iterations; i++) {
    for (int p = 0; p < UNROLL; p++) {
        for (int j = 0; j < DLP; j++) {
            mulR[j]*=base;
            addR[j]+=1;
            addR[DLP+j]+=1;
        }
    }
}
```

We optimized the value for UNROLL and DLP in the range of 1 to 19 and found the following results:

Operation	Instruction Set	DLP	UNROLL	Performance [Flop/Cycle]	
				Measured	Theoretical
ADD	x87	9	14	0.965	1.0
	SSEScalar	7	15	0.934	
	SSE	7	15	0.986	
MUL	x87	3	10	0.498	0.5
	SSEScalar	3	9	0.498	
	SSE	3	10	0.498	
Mixed	x87	3	17	1.31	1.5
	SSEScalar	3	17	1.35	
	SSE	2	4	1.34	

For pure additions and multiplications we almost reached the theoretical limit. For the mixed workload we reached 90% of the theoretical 1.5 flops per cycle.

In our plots we show 1 flop per cycle as single core performance ceiling and 2 flops per cycle as dual core performance ceiling.

The bandwidth ceiling can be calculated from the parameters of the memory subsystem and is 2.80 bytes per cycle (see section 2.2).

To measure the bandwidth which can actually be reached we implemented a micro benchmark. To measure the read performance, the benchmark iterates over a buffer and combines the data using the XOR operation. For the write performance a memory buffer is overwritten.

We use SSE load and stores. There are two flavors of the store operations. Normal stores first load the data from memory into the caches and overwrite it there. When the cache line gets evicted, the data is written to memory. Non temporal stores do not load the data into the caches. The data is written directly to memory.

We use multiple load and store streams. That is, we split the buffer into multiple parts and iterate over all parts simultaneously.

We experimented with software prefetching but could not improve our results.

Optimizing loop unrolling and stream count resulted in the following numbers:

Operation	Streams	UNROLL	Throughput [Byte/Cycle]
Load	2	3	2.16
Write	3	4	1.71
Non Temporal	1	1	1.66

The throughput of the non temporal stores seems to be low compared to the normal stores. But since only half of the memory has to be transferred (no loads), the same amount of memory is actually written in half the time.

The results reported here were obtained using MemBus, but the results of the MemL2 variant are almost identical.

In the stream benchmark [5] the system reaches a copy rate of 2434 MB/s, which is $2 * 3434MB/1.8GHz = 2.84$ bytes per cycle. Our micro benchmark reported a smaller number, but the result is comparable to ours.

We also measure the random access throughput. A large buffer is allocated and randomly accessed. The throughput is 0.525 bytes per cycle. Since a cache line is 64 bytes, this results in one memory access every $64/0.525 = 122$ cycles.

In our roofline plots we only show the load ceiling (2.2 flops per cycle) and the random access ceiling (0.52 bytes per cycle).

Chapter 6

Experimental Results

We used our tool to measure various kernels. For each kernel we describe the kernel and the measurement setup and show the resulting roofline plot.

We used the techniques described in sections 3.4 and 4.4 to reduce the variance of the measurement results.

We repeated each measurement 10 times and show the results using whisker bars for the minimum and maximum and error boxes for the 25th and 75th percentile.

Since the precision generally improves with growing problem sizes, we decided to measure large problems only once. For such measurements, no whisker bars or error boxes are drawn.

To investigate the influence of the initial cache state, we measured a specific implementation of a kernel with cold caches, with the data loaded into the caches ('-Data'), with the code loaded into the caches ('-Code') and with both data and code loaded into the caches ('-Data-Code').

We observed a similar behavior for all kernels. If only the code is initially loaded into the caches, the operational intensity is slightly increased for small problem sizes, since the transfer volume is reduced by the size of the code. For larger problem sizes the effect disappears, since the size of the code is negligible to the transfer volume needed to load the data.

If the data is initially loaded into the caches, the operational intensity is increased compared to cold caches and grows with the size of the vector. During the measurement the code has to be loaded once from memory. Since the data is already in the caches the transfer volume is constant, while the operation count grows with the problem size.

With growing problem sizes the data is too large to fit into the caches. The operational intensity drops to the level of cold caches. The drop should occur when the size of the data exceeds the size of the caches.

If both data and code are initially loaded into the caches, the operational intensity is theoretically infinite (or undefined) as long as both data and code fit into the caches, since no memory transfer is required. With growing problem sizes the data does not fit into the caches anymore and the operational intensity drops but stays higher than the operational intensity measured when only the data is loaded into the caches.

For high operational intensities we observed a reduced precision. Since such measurements run for a relatively long period of time while causing a small transfer volume, auxiliary data transfers might be measured which occur at a fluctuating rate. This could cause the loss of precision.

The effect on performance of the initial cache state is generally smaller than the effect on operational intensity. While a factor of 10 is not uncommon for the operational intensity, an increase by 1.5 can only be observed for memory bound kernels. The effect is marginal for computationally bound kernels.

6.1 BLAS

Basic Linear Algebra Subprograms (BLAS) is a standard API for basic linear algebra operations. The functionality is divided into three levels:

Level 1 Vector-Vector Operations

Level 2 Matrix-Vector Operations

Level 3 Matrix-Matrix Operations

For each level, we chose a representative operation and generated roofline plots using the Intel Math Kernel Library (MKL) [11] and the OpenBlas [3] implementations.

6.1.1 Level 1

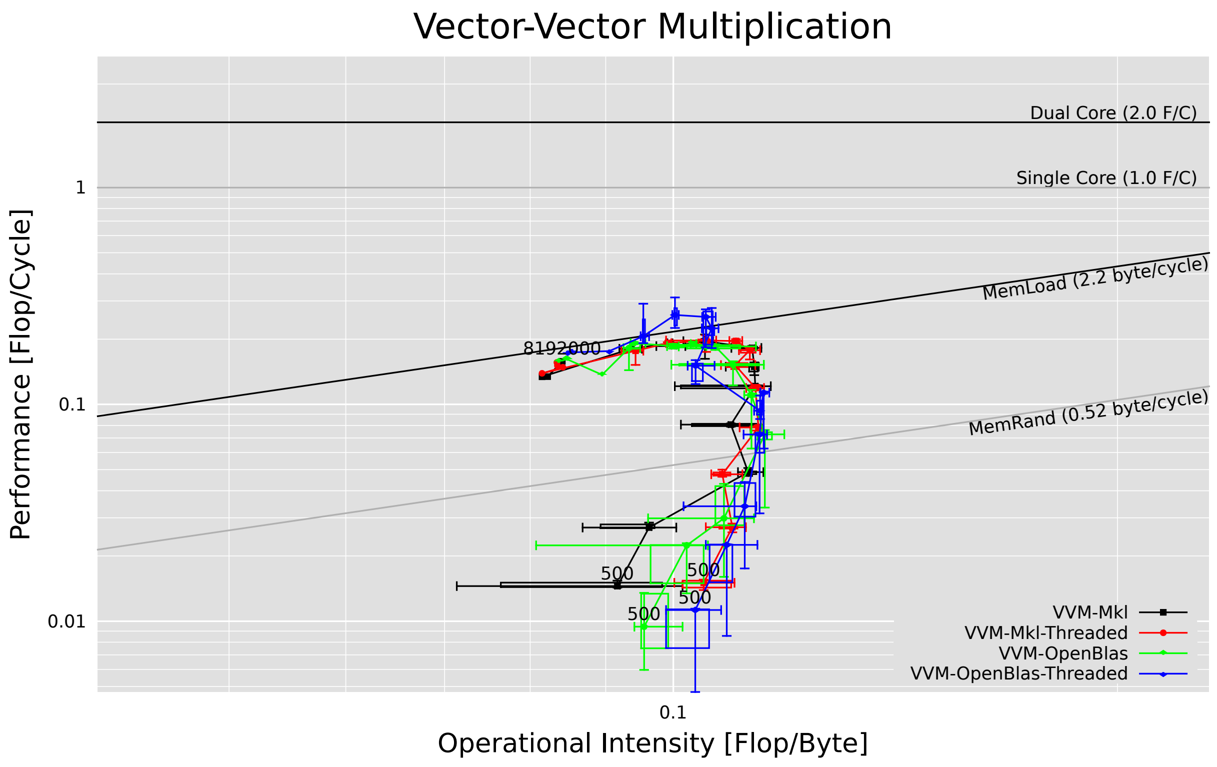


Figure 6.1: Roofline Plot of the Vector-Vector Multiplication (MemL2, DoublePrecisionFlop)

For level 1, we chose the 'daxpy' operation, defined as $\mathbf{y} \leftarrow \alpha\mathbf{x}+\mathbf{y}$ with double vectors. There is no reuse in this kernel. For every vector element triple, two operations are performed. If the vectors fit into the caches there is no write back and the operational intensity is $2/(2*8) = 1/8 = 0.125$. For larger vectors the intensiy becomes $2/(3*8) = 1/12 = 0.083$. These values are confirmed by figure 6.1. The precision of the results is good.

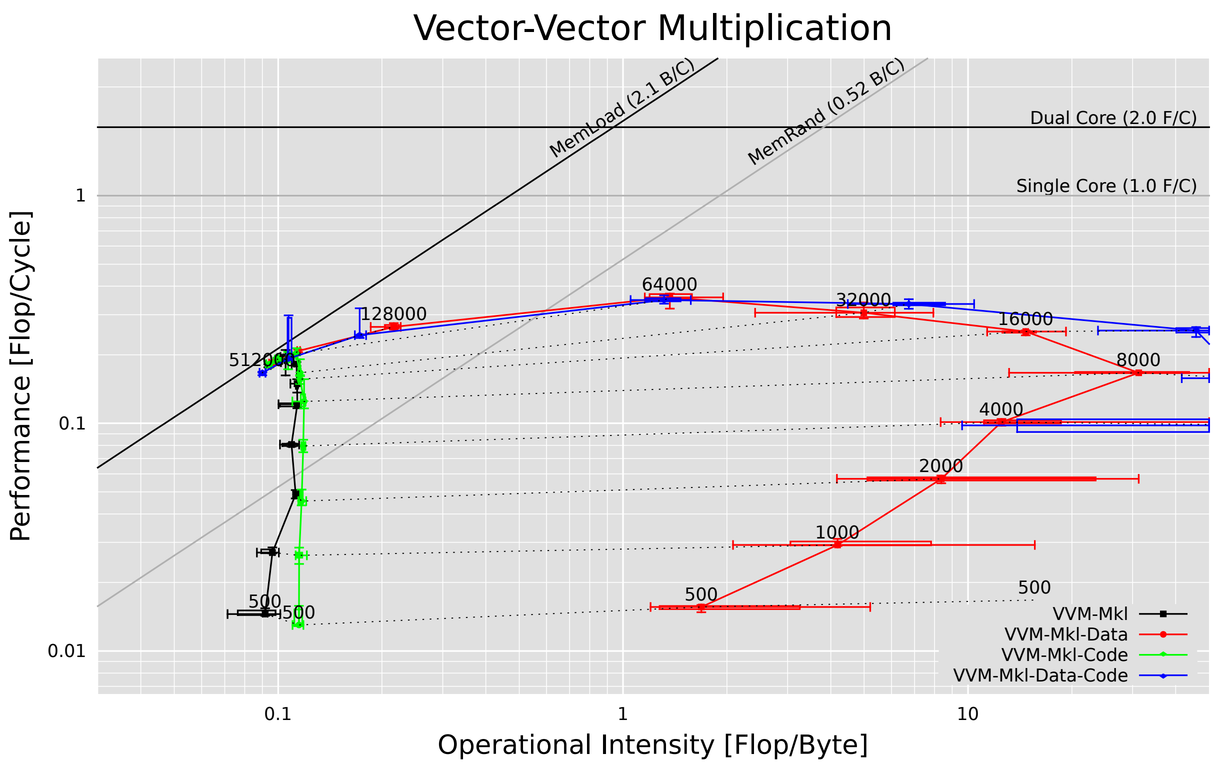


Figure 6.2: Roofline Plot of the influence of the initial cache state on the Vector-Vector Multiplication (MemL2, DoublePrecisionFlop)

If data or code and data is initially loaded into the caches, we expect the operational intensity to drop for a vector size of $n = 2MB/(2*8) = 131'072$. But the drop already starts at a size of 16'000 (fig. 6.2). The two double vectors of this size occupy one eight of the cache. Since the L2 cache is 8 way associative, this is exactly the size which fits into the cache without using one cache line set twice. We did not investigate this further.

6.1.2 Level 2

For level 2, we chose the 'dgemv' operation, defined as $\mathbf{y} \leftarrow \alpha A\mathbf{x}+\beta\mathbf{y}$ with double vectors and matrices. For an $n \times n$ matrix and vectors of size n, the operation count is $2n$ for the vector scaling and $2n^2$ for the multiplication of the matrix with the vecor and the vector addition, totaling in $2n^2+2n$. The vector $\mathbf{x}$ can be reused. Since the vector sizes we measure fit into the last level cache, the write backs are cached. The memory transfer is $2n*8$ bytes to load the vectors and n^2*8 bytes to load the matrix. This results in a maximal operational intensity for large matrices of $2/8 = 0.25$ which is confirmed by figure 6.3.

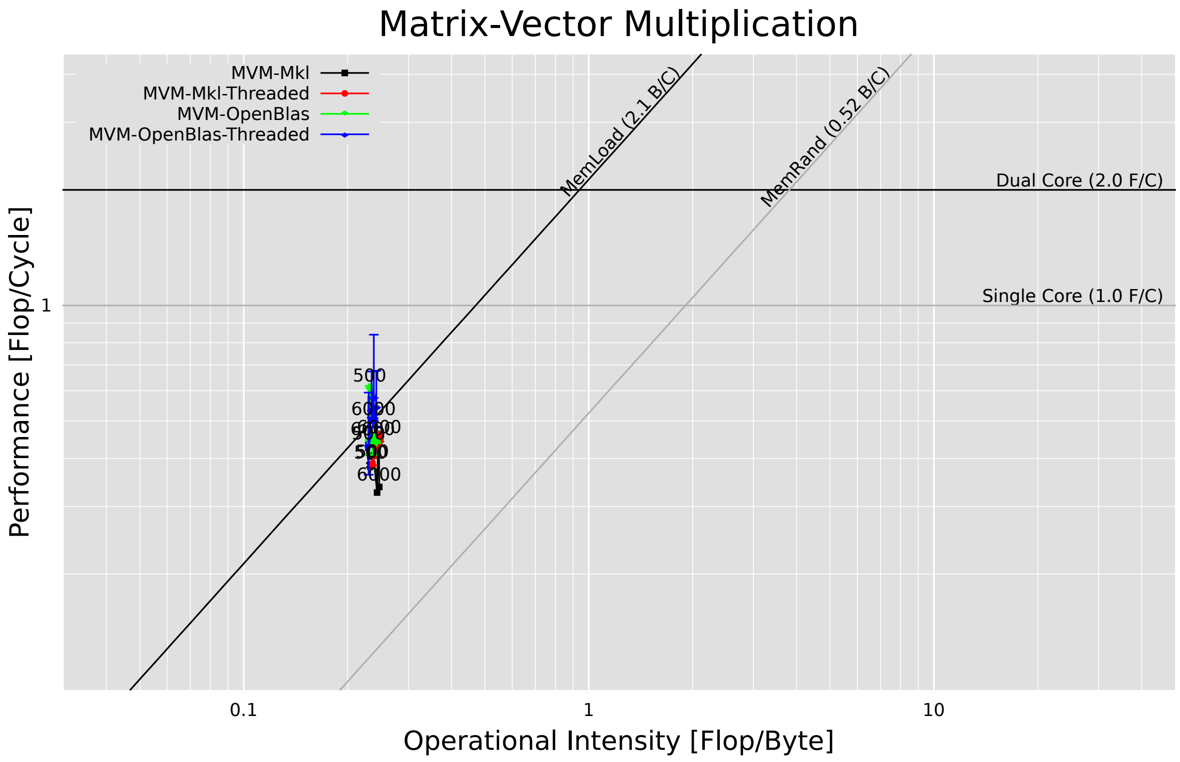


Figure 6.3: Roofline Plot of the Matrix-Vector Multiplication (MemL2, Double-PrecisionFlop)

The precision is generally good, although we observe some performance outliers for the multi threaded OpenBlas implementation.

When initially loading data or code and data into the caches, the operational intensity starts to drop at a vector size of $n = 300$. At this point the matrix occupies $300^2 * 8 = 720000$ bytes, which is about one third of the cache.

6.1.3 Level 3

For level 3, we chose the 'dgemm' operation, defined as $C \leftarrow \alpha AB + \beta C$ with double matrices. For an $n \times n$ matrix, the operation count is $2n^2$ for the two scale operations and $2n^3$ for the matrix-matrix multiplication and the addition to C.

The minimal memory transfer volume is $3n^2 * 8$ bytes to load the matrices and $\max(n^2 * 8 - k, 0)$ bytes to store the result (write back caching), where k is the size of the last level cache. In our measurement system k is 2MB.

The maximal operational intensity is $(2n^3 + 2n^2)/(3n^2 * 8 + \max(n^2 * 8 - k, 0))$. For large n this becomes $(2n^3)/(4n^2 * 8) \approx n/16$. This operational intensity cannot be reached since reuse is bound by the cache size. The intensity achieved in practice depends on the algorithm. In the following paragraphs we will analyze the triple loop and the blocked version. To simplify matters, we will ignore the effect of code and small parts of the working set on the cache.

The triple loop uses the following algorithm:

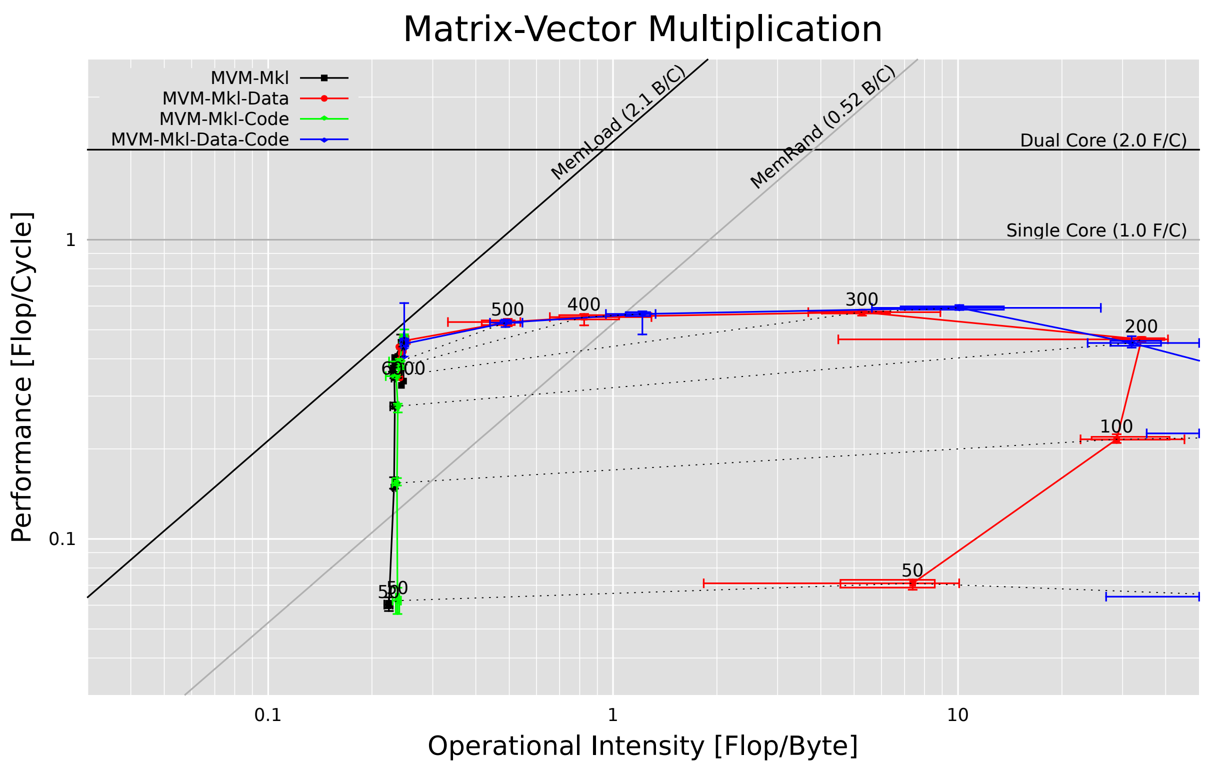


Figure 6.4: Roofline Plot of the Matrix-Vector Multiplication (MemL2, Double-PrecisionFlop)

```
for (long i = 0; i < size; i++)
for (long j = 0; j < size; j++)
for (long k = 0; k < size; k++)
c[i * size + j] += a[i * size + k] * b[k * size + j]
```

If B fits into the cache, all matrices have to be loaded exactly once. In this case the transfer volume is $3n^2 * 8$ to load the matrices. Since the space occupied by the reused matrix is not available for write back caching, $\max(n^2 * 8 - \max(k - n^2 * 8, 0), 0)$ bytes are written. Dropping lower order terms, the operational intensity is optimal $(n/16)$.

As soon as B does not fit into the cache $(n^2 * 8 > k; n > 512)$ it is loaded over and over. In addition, this trashes the the write back cache. The resulting memory transfer volume is $(n^3 + 3n^2) * 8$ and, for large matrices, the operational intensity is 1/4.

For very large matrices, the lines of A do not fit into the cache $(n * 8 > k; n > 262144)$.

The blocked version uses the following algorithm:

```
for (long i = 0; i < size; i += Nb)
for (long j = 0; j < size; j += Nb)
for (long k = 0; k < size; k += Nb)

for (long id = i; id < i + Nb; id += Mu)
for (long jd = j; jd < j + Nb; jd += Nu)
for (long kd = k; kd < k + Nb; kd += Ku)
for (long kdd = kd; kdd < kd + Ku; kdd++)
for (long idd = id; idd < id + Mu; idd++)
for (long jdd = jd; jdd < jd + Nu; jdd++)

c[idd * size + jdd] +=
  a[idd * size + kdd] * b[kdd * size + jdd];
```

We chose $N_b = 50$ and $N_u = M_u = K_u = 2$. For our discussion, only the blocking with Nb is relevant. It is presumed that N_b divides n.

If the matrix B fits completely into the cache ($n^2 * 8 < k; n < 512$), we again have a memory transfer volume of $3n^2 * 8 + \max(n^2 * 8 - \max(k - n^2 * 8, 0), 0)$.

The blocking divides the matrices into block lines and block columns, each containing $\frac{n}{N_b}$ blocks.

If the cache cannot hold the whole matrix but a whole block line fits into the cache ($\frac{n}{N_b}N_b^2 * 8 = nN_b * 8 < k; n < 5243$), the blocks of A are reused. Thus A is loaded once while each block of B is loaded $\frac{n}{N_b}$ times, once per block line of A. The memory transfers due to the loads is $(2n^2 + \frac{n}{N_b}n^2) * 8 = (2n^2 + \frac{n^3}{N_b}) * 8$. The effect of outstanding writes can be neglected. The writes cause an additional $n^2 * 8$ bytes to be transferred. Dropping lower order terms, the operational intensity becomes $2n^3/(n^3/N_b * 8) = N_b/4 = 12.5$

For large matrices the block lines of A do not fit into the cache ($n > 5243$). Each block of A is loaded once per block column of B and each block of B is loaded once per block row of A. Thus the memory transfer to load A and B is $2\frac{n}{N_b}n^2 * 8 = 2\frac{n^3}{N_b} * 8$ and $2 * n^2 * 8$ to load and store C. The resulting operational intensity is $N_B/8 = 6.25$.

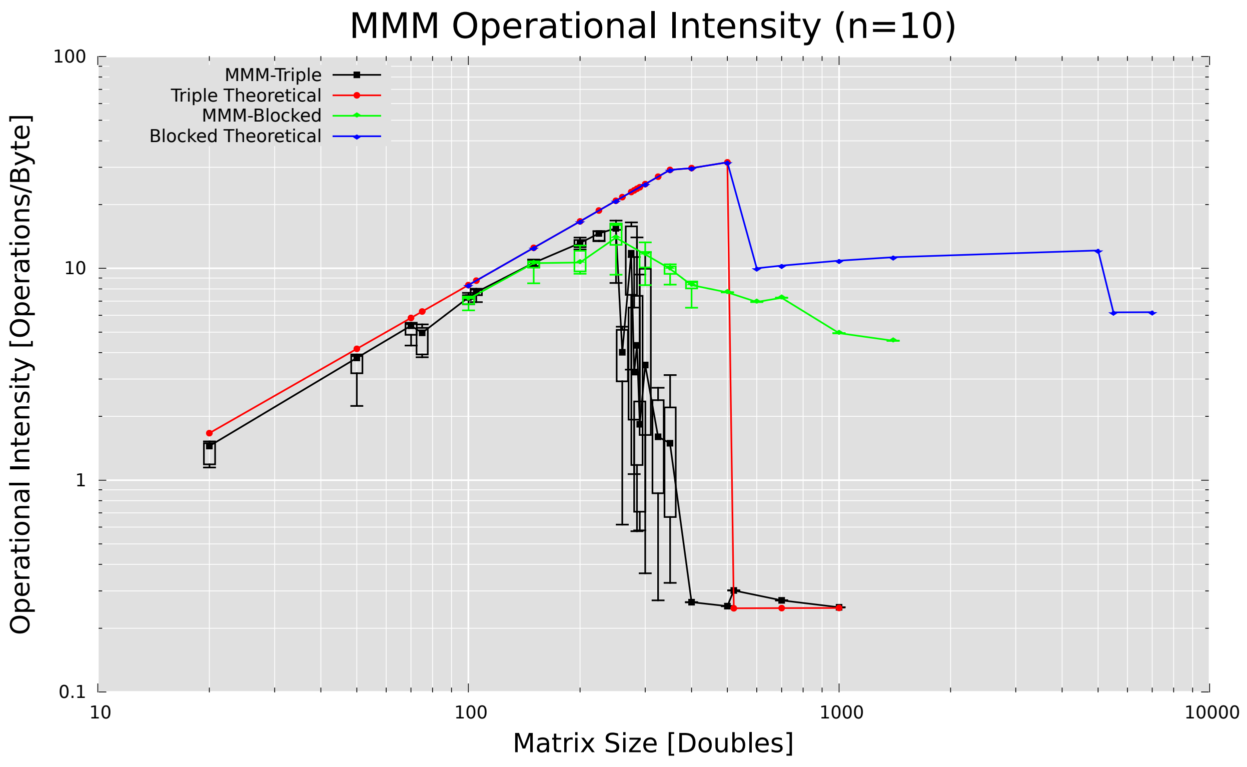


Figure 6.5: Matrix Matrix Multiplication: Operational Intensity (MemL2)

When comparing these models for the operational intensity of the triple loop and the blocked version, we observe a good match for matrix sizes up to $n = 200$. Past this size, the operational intensity of the triple loop rapidly falls to the value predicted for large matrices, but the drop begins too early. We predicted the fall when a single matrix occupies more memory than the L2 cache size, but the measured drop begins at half the cache size. (fig 6.5).

The model for the blocked version overestimates the measurement results as well. The initial drop begins too early. The operational intensity for large matrices is about half the predicted value.

In attempt to find the reason for the observed behavior we analyzed the TLB misses. The TLB has 128 entries. For the triple loop we predict $3 * \frac{n^2 * 8}{4KB}$ TLB misses as long as one matrix fits into 128 pages ($\frac{n^2 * 8}{4KB} < 128; n < \sqrt{128 * 4KB/8} = 256$). If the matrices are larger, calculating a single element of the result requires reading one element of every row of B. If the rows of B are smaller than one page, multiple accessed elements will lie in the same page. If the rows are larger, one page is accessed per row. Thus the number of pages accessed to compute one element of the result is $\min(n, n^2 * 8/KB)$. Since the TLB is trashed during accessing a column of B, there are $n^2 \min(n, n^2 * 8/KB)$ TLB misses to compute the n^2 result elements, ignoring the TLB misses due to accessing A and C.

For the blocked version, we provide an analysis for a very restricted range only. If one line of a matrix is larger than one page ($n * 8 > 4KB; n > 512$), each row within a block will lie in it's own page. If each block line fits within one page ($N_b * 8 < 4KB; N_b < 512$) and the TLB is trashed during each block multiplication, there will be $3 * N_b$ TLB misses per block multiplication, $3(\frac{n}{N_b})^3 N_b = 3\frac{n^3}{N_b^2}$ in total.

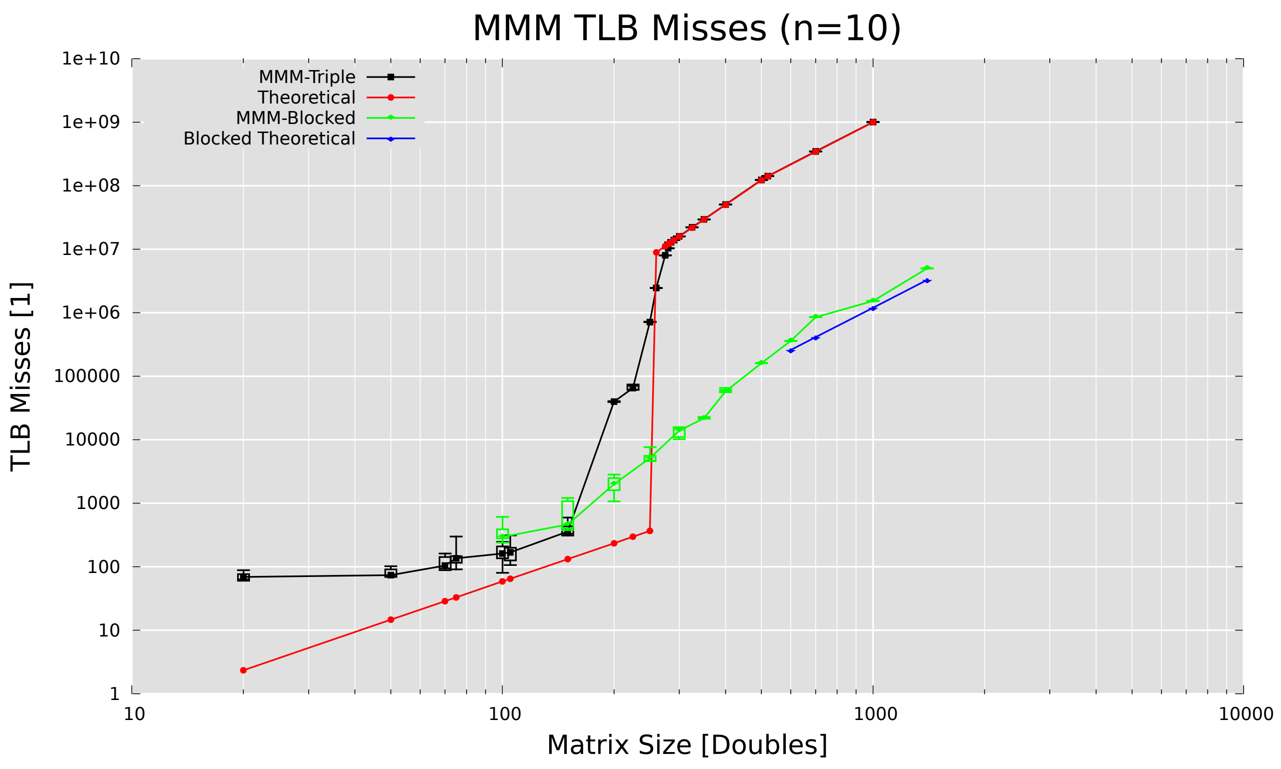


Figure 6.6: Matrix Matrix Multiplication: TLB Misses

Figure 6.6 shows that the measurement results follow our predictions.

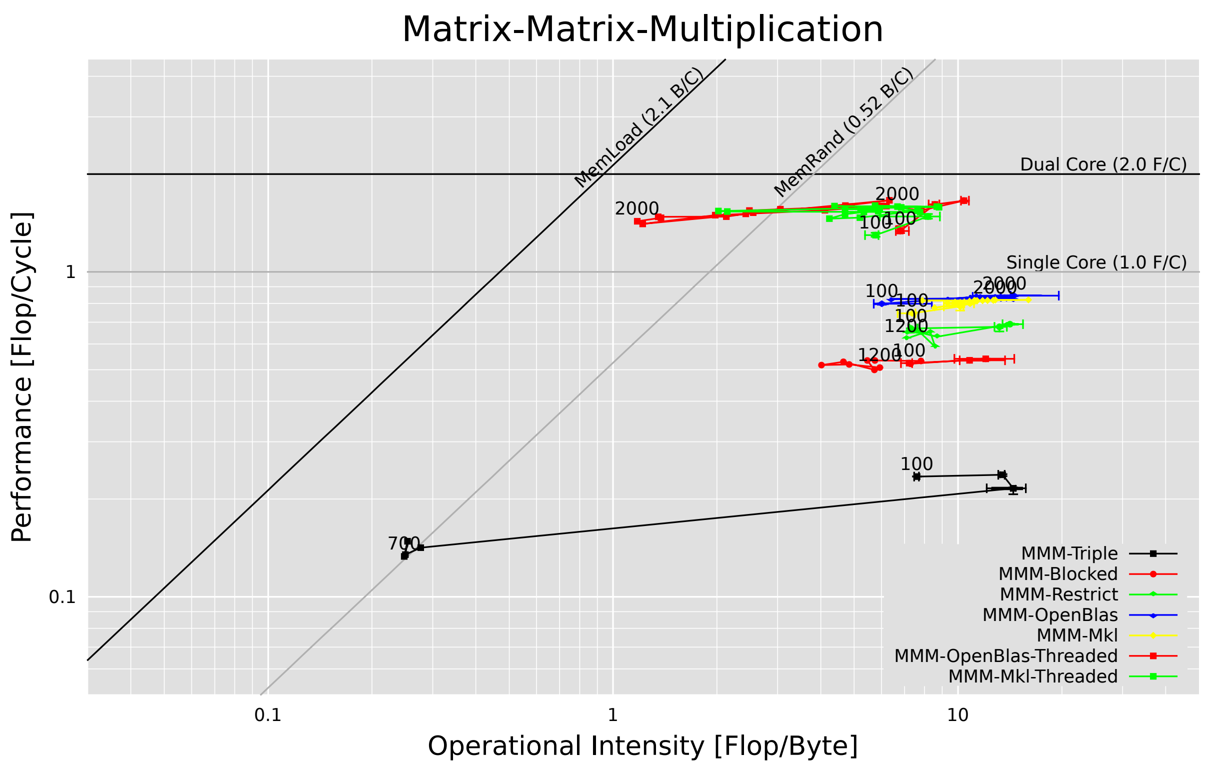


Figure 6.7: Roofline Plot of the Matrix-Matrix Multiplication (MemL2)

Figure 6.7 shows the roofline plot of the triple loop, the blocked version and the two libraries.

In addition we included the blocked version with scalar replacement. The scalar replacement has been implemented by specifying the matrices as 'restrict' pointers to the compiler, hence the name.

The precision of all results is decent.

The triple loop has a poor performance, even for small matrices. This is probably due to it's lack of unrolling. For matrix sizes exceeding $n = 400$ the drop in operational intensity and performance is clearly visible.

The other single threaded implementations all have an operational intensity of around 10. The performance is 0.5 flops/cycle for the blocked version, 0.7 for the blocked version with scalar replacement and around 0.85 for the library implementations. It is interesting to see that the highly optimized library have a very similar operational intensity and only a 20% higher performance compared to the rather similar blocked version with scalar replacement.

The multi threaded implementations show a speedup of factor two, with a reduction in operational intensity. It would be interesting to know if the reduced operational intensity is a measurement artifact or due to the implementations.

The initial cache state has a very small influence on the performance. (fig. 6.8) This results in a very cluttered roofline plot. The insensitivity of the performance to the initial cache state is plausible since the matrix-matrix multiplication has a relatively high operational intensity and is not memory bandwidth bound.

To be able to analyze the effect of the initial cache state on the operational intensity, we extracted the relevant information in figure 6.9.

For matrix sizes greater than $n = 300$ the initial cache state has almost no

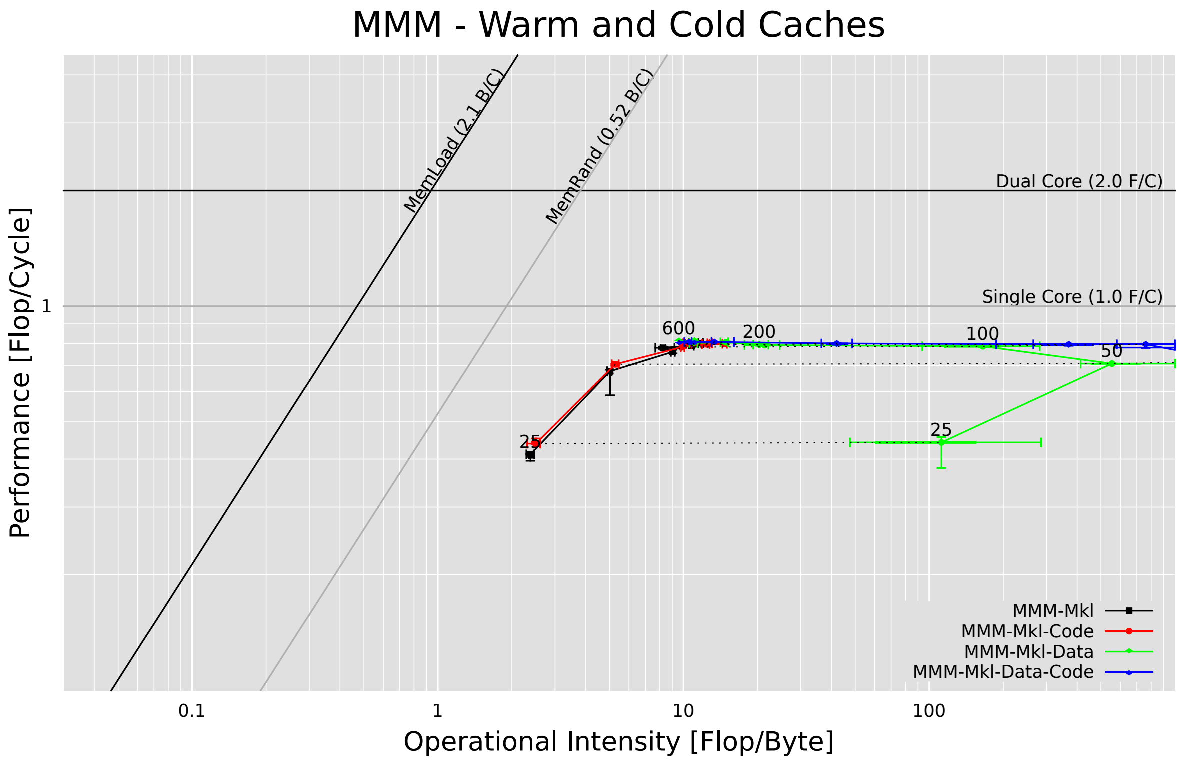


Figure 6.8: Roofline Plot of the Matrix-Matrix Multiplication with Different Initial Cache States (MemL2)

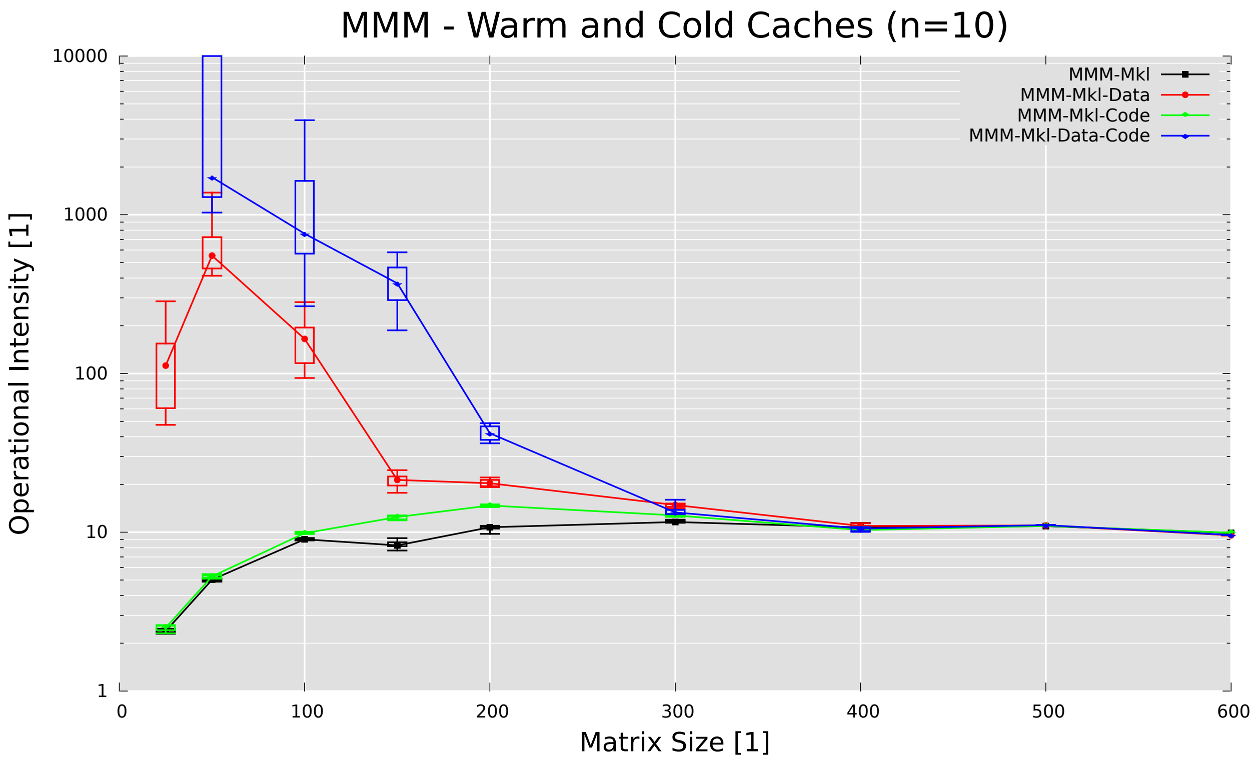


Figure 6.9: Effect of the Initial Cache State on the Operational Intensity for the Matrix-Matrix Multiplication (MemL2)

effect.

The effect of loading the code into the caches is negligible for small matrix sizes. Starting with $n = 150$ the difference becomes apparent. This could be caused by a switch of the algorithm implementation by the library.

If the data is initially loaded into the caches the operational intensity increases by more than an order of magnitude for small matrices. The effect grows with larger matrices. But for a size of $n = 100$ the operational intensity begins to drop. One matrix of that size occupies 80KB only. We have no explanation for this effect.

At $n = 150$ the drop stops, with an operational intensity still two times larger than the one for cold caches. The effect of the warm caches decays for growing matrix sizes.

If both data and code is loaded into the caches we observe operational intensities in the range of 200 to 10000 for small matrix sizes. We see a similar drop curve as when only the data is loaded into the caches.

The huge operational intensities can be explained by the large number of operations that are performed while theoretically not transferring any memory. The operation count for $n = 25$ is $2 * n^3 = 31'250$ and $250'000$ for $n = 50$.

6.2 FFT

Fast Fourier Transform algorithms 'are of great importance in a variety of fields, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.' ([20])

Apart from the simple implementation found in 'Numerical Recipes' (NR) [13] on page 608, we used the implementations from the MKL [11] , FFTW [1] and Spiral [4]. All four implementations operate on a linear buffer containing n complex numbers, each represented by two doubles. The transformation is performed in-place.

Figure 6.10 shows the roofline plot of the four implementations. The NR implementation performs well initially, but the operational intensity and the performance significantly drop for large input sizes. The library implementations have a large overhead on small input sizes, but perform a lot better on large inputs.

Figure 2.1 shows the influence of the initial cache state. When data or code and data is initially loaded into the caches, the operational intensity drop starts for $n = 4'096$. The input size is $n * 2 * 8 = 64KB$.

6.3 WHT

The Walsh-Hadamard transformation (WHT) is related to the FFT. It's application is mainly in the field of data encoding.

We used the implementation of the SPIRAL project [4]. The algorithm operates on a buffer of 2^n doubles. Since we only have one implementation, we directly show a roofline plot with the different initial cache states (fig. 6.11)

Since the roofline plot is very cluttered, we show the operational intensity versus the problem size in figure 6.12. For warm data or code and data, the operational

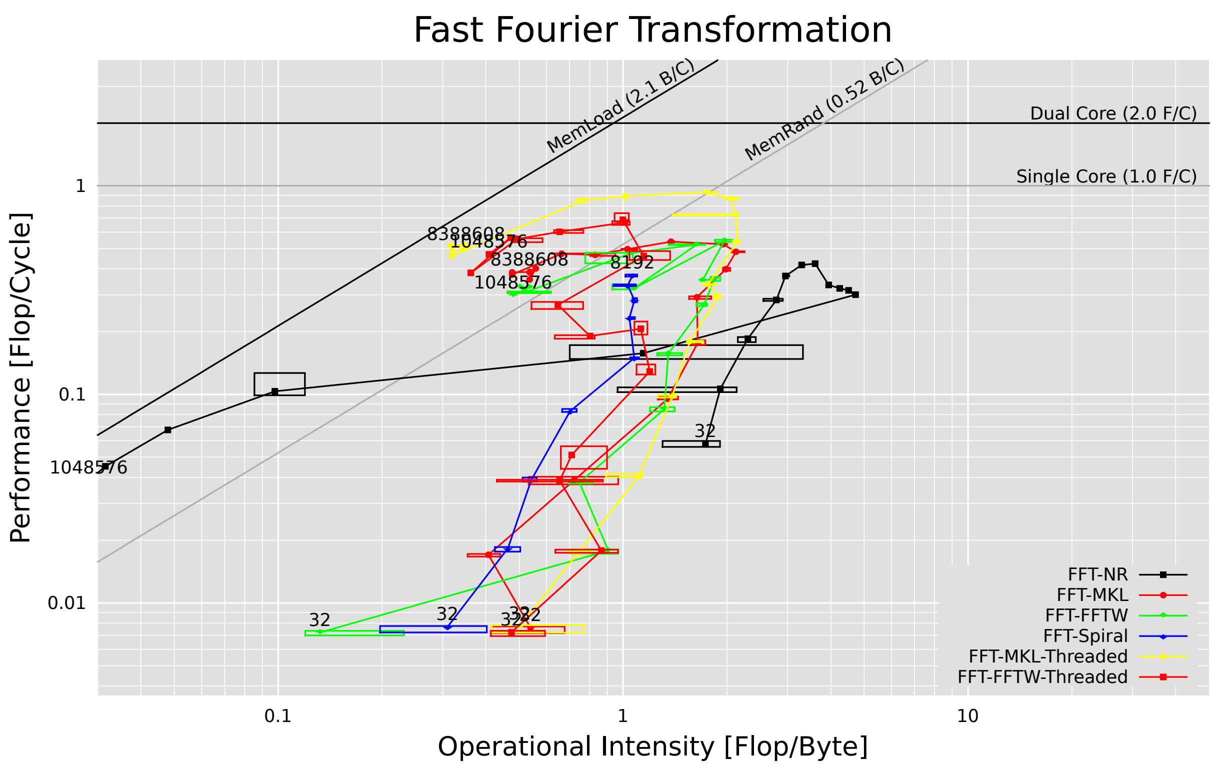


Figure 6.10: Roofline Plot of four FFT implementations (MemL2)

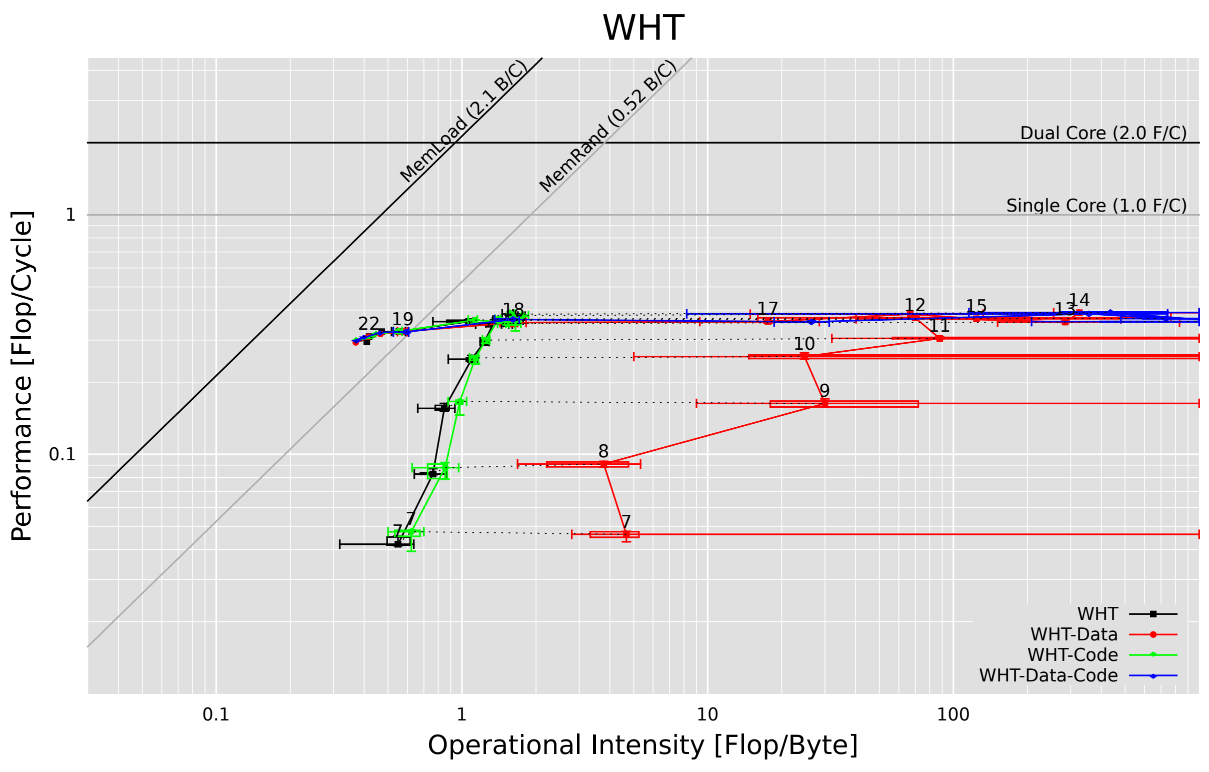


Figure 6.11: Roofline Plot WHT (MemL2)

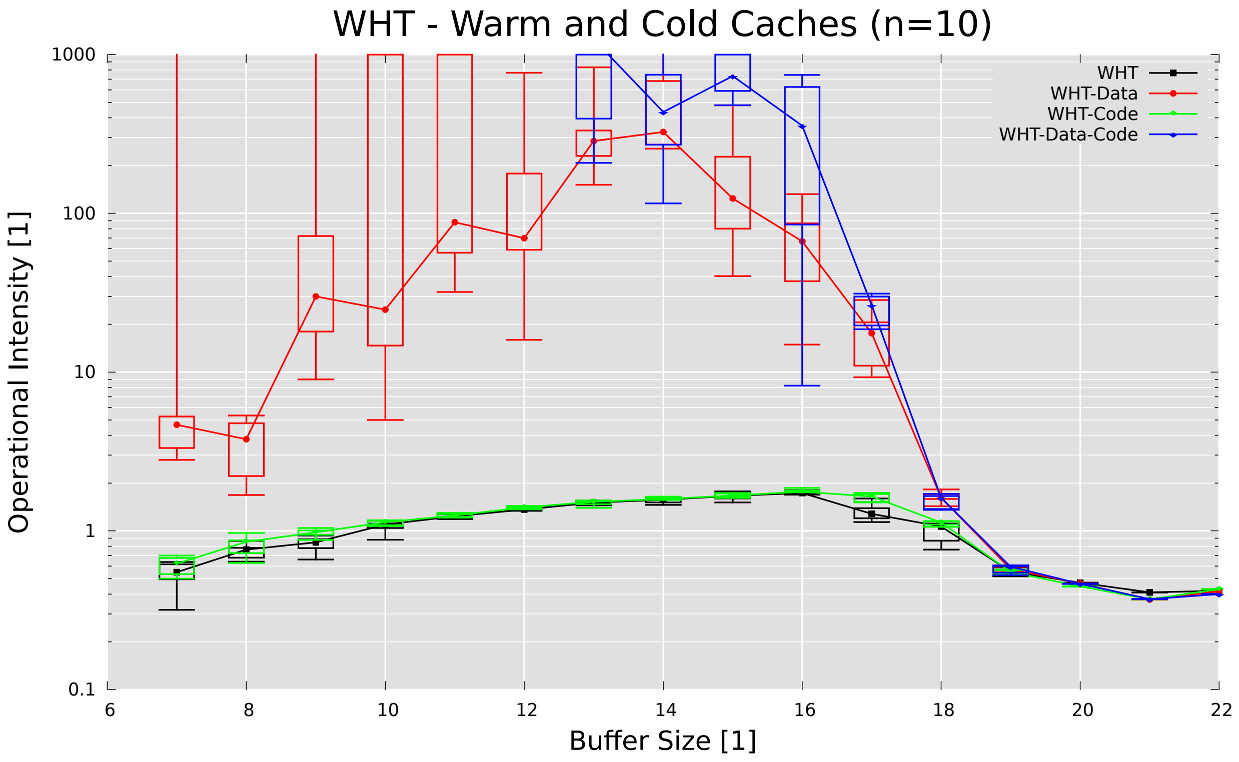


Figure 6.12: Operational Intensity WHT (MemL2)

intensity drops at $n = 15$. The buffer size is $2^n * 8 = 262'144$, which is one eight of the second level cache. We have no explanation for this effect.

Chapter 7

Future Work and Conclusions

In this thesis we designed and implemented a tool which simplifies performance measurements. We introduced multiple abstractions, which allow simple and common measurements to be performed with ease, while being flexible enough to scale to more complex situations.

Due to these abstractions it should be possible to repeat the measurements presented in this thesis on different systems with small effort. We already repeated the measurements on an Intel Core machine (see appendix C). To reach more potential users it would be good to port the tool to more architectures.

The precision and accuracy of our measurement results are satisfactionary for the generation of roofline plots. However, we observe substantial variations when measuring memory intensive kernels. We have no good explanation for these effects. A thorough analysis could be very informative.

In various places we observed cache misses when all data should theoretically fit into the L2 cache. We suspect that the LRU (least recently used) cache line replacement implementation is not perfect. Detailed experiments could possibly tell if this is true or false.

For handling the variance we presumed that a measurement either results in the true value, or it is disturbed by some activity (task switches) and results in a biased result. We further presumed that the distortion can only increase the result (time, memory transfer, operation count), and thus the minimum result of multiple measurement runs has to be the true result.

An in depth study of this model could result in means to reduce the number of measurements to be performed and the ability to give confidence intervals for measurement results.

Appendix A

Measurement Tool Architecture

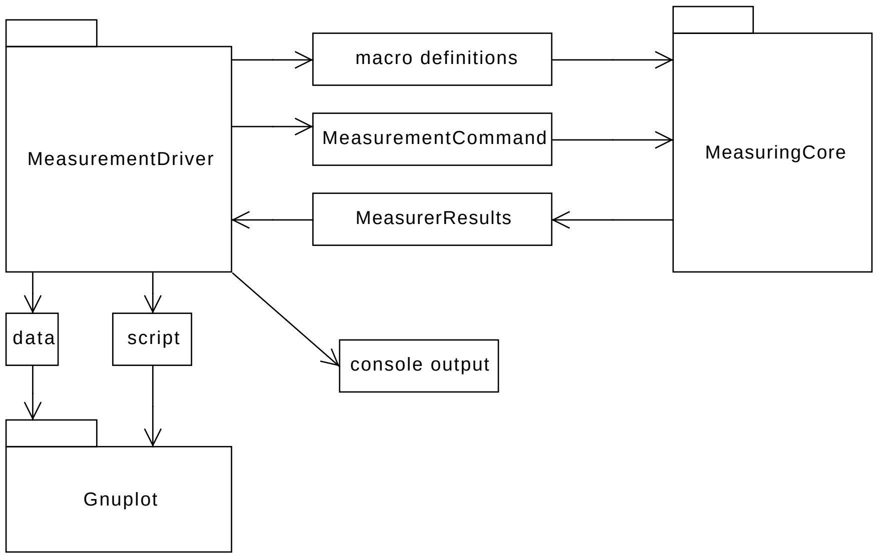
The tool consists of two main components: The Measuring Core, which performs the actual measurements, and the Measurement Driver, which controls the core.

High performance code is generally written in C or C++. Therefore the core is written in that language. To simplify development and maintenance, we tried to keep the amount of C++ code as small as possible. Therefore the measurement driver is written in Java.

A measurement is controlled by a measurement specific routine (one per measurement), which iterates through all parameter points to be examined. For each point it compiles and starts the Measuring Core and processes the results. Since these routines tend to require a lot of flexibility, we decided to use Java for this code.

During the development of measurement control routines, it is expected that many changes do not affect the parameter points. To speed up repeated measurements after changes to the measurement driver, the measurement results are cached. Thus, as long as the measurement parameters are not changed, the measurement does not need to be repeated.

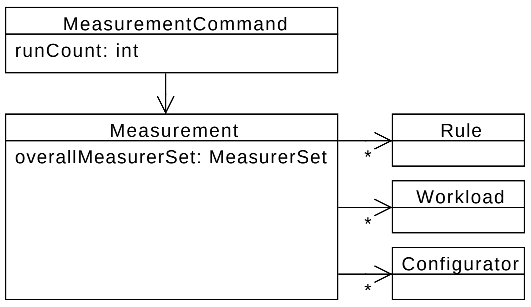
The measurement tool is used to generate and display measurement results. Often, the measurement results lead to changes to the tool itself. Thus, switches between using the tool and developing the tool are frequent. To support these switches, a frontend program is provided. It compiles the measurement driver and executes it. It can be started using a shell script called ”rot”. The result files of a measurement are placed in the current working directory.



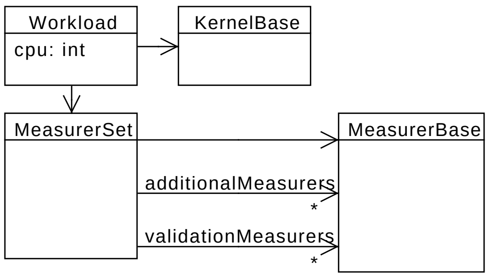
A.1 Component Collaboration

Data transfer between the measuring core and the measurement driver is achieved using serialized objects stored in files. Classes of these objects are used both from C++ and Java. They are called shared entities and are described using XML. Source code for both languages is generated. For details, see A.3

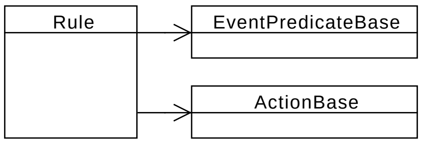
The root class for describing a measurement is a MeasurementCommand. It contains the number of times the measurement should be repeated, as well as the actual Measurement. The kernels are contained within the workloads, and rules allow to respond to various events happening during the measurement.



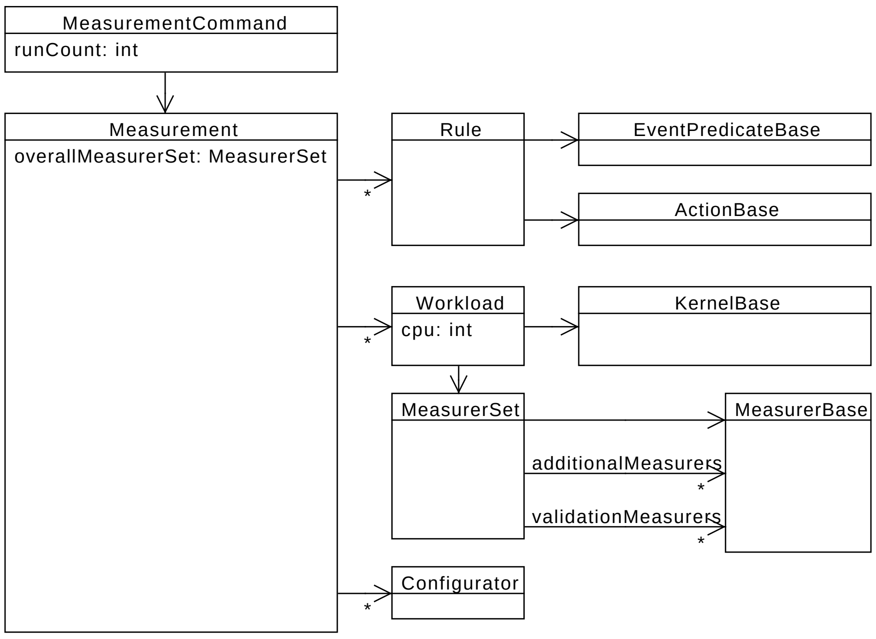
A workload describes what should be run and measured on one core. Each workload is run within a separate thread, which is optionally pinned to a fixed core. In this thread, the validation measurers are started, the caches are warmed up, the additional measurers are started, followed by the main measurer. Then the kernel is run and the measurers are stopped.



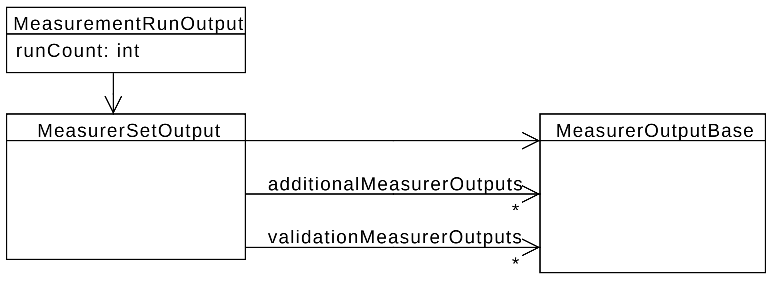
During the measurement, events are raised. For example: start of a workload, end of a workload, start of a thread etc. These events are matched against the event predicates stored in the rules. If a predicate matches, the action of the rule is executed.



The following diagram shows all classes describing a measurement together:

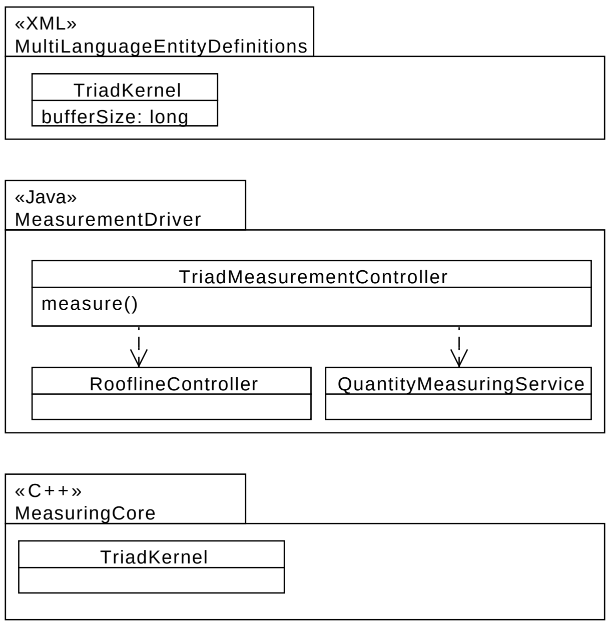


To be able to analyze the distribution of the results, a measurement is tipically repeated multiple times. Each repetition is called measurement run. In each run, the outputs of all measurers are collected. At the end of the measurement, the core serializes the results of all runs into a single file, which is read by the driver.



A.2 Measurement Overview

In this section, we will look at the components specific to a measurement. The following diagram gives an overview of the discussed components:



First, we will look at the kernel. It is defined in an XML file:

```
   <?xml version="1.0" encoding="UTF-8"?>
 2 <derivedClass
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4     xsi:noNamespaceSchemaLocation="../shared.xsd"
       name="TriadKernel" <!-- name of the class -->
 6     baseType="KernelBase"
       cSuffix="Data"
 8     comment="Kernel performing a=b+k*d
           on a memory buffer">
10     <field
           name="bufferSize"
12         type="long"
           comment="The size of the buffer"/>
14 </derivedClass>
```

Kernels are always derived from KernelBase, hence the ‘derivedClass’ element on line 2 and the base type defined on line 6.

The ‘cSuffix’ is given as ‘Data’ on line 7. This causes the generated class to be named ‘TriadKernelData’. The measuring core implements ‘TriadKernel’, which derives from TriadKernelData. The serialization service will instantiate the derived class. This mechanism allows to use a derived class, optionally with additional code and data, to be used in the measuring core. This is how the actual algorithm is implemented. We will look at this later.

On line 10 starts a field definition. Fields and getters/setters are generated for the field.

Next comes the class controlling the whole measurement.

```
package ch.ethz.ruediste.roofline.measurementDriver.measurementControllers;

public class TriadMeasurementController implements IMeasurementController {

    public String getName() {
        return "triad";
    }

    public String getDescription() {
        return "runs the triad kernel";
    }

    @Inject
    public QuantityMeasuringService quantityMeasuringService;

    @Inject
    public RooflineController rooflineController;

    public void measure(String outputName) throws IOException {
    ...
    }
}
```

The class implements **IMeasurementController** and has to be placed in the ‘measurementControllers’ package. The measure command will instantiate the class and call the measure() method. The getName() method returns the name of the measurement, which is used to identify the measurement.

The two fields with the @Inject attribute are initialized by the dependency injection framework when the class is instantiated. The quantity measuring service allows to measure quantities like operation count, transferred bytes, performance etc. The roofline controller manages a roofline plot. We will see how these facilities are used when we look at the body of the measure function:

```
public void measure(String outputName){
    // initialize the roofline plot
    rooflineController.setTitle("Triad");
    rooflineController.addDefaultPeaks();

    for (long size = 10000; size < 100000; size += 10000) {
        // initialize kernel
        TriadKernel kernel = new TriadKernel();
        kernel.setBufferSize(size);
        kernel.setOptimization("-O3");

        // add a roofline point
        rooflineController
                .addRooflinePoint("Triad", size,
                        kernel, Operation.CompInstr,
                        MemoryTransferBorder.LlcRamBus);

        // create calculators
        QuantityCalculator<Throughput>
           throughtputCalculator
            = quantityMeasuringService
                .getThroughputCalculator(
                    MemoryTransferBorder.LlcRamBus,
                    ClockType.CoreCycles);

        QuantityCalculator<OperationCount>
           operationCountCalculator
            = quantityMeasuringService
                .getOperationCountCalculator(Operation.
                   CompInstr);

        // perform measurement
        QuantityMap result = quantityMeasuringService.
           measureQuantities(
                kernel, throughtputCalculator,
                   operationCountCalculator);

        // print throughput and operation count
        System.out.printf("size %d: throughput: %s
           operations: %s\n", size,
                result.best(throughtputCalculator),
                result.best(operationCountCalculator));
    }

    // create the PDF of the plot
    rooflineController.plot();
}
```

First the roofline plot is initialized with the title and the default peaks. Then, for each buffer size, the kernel is initialized. For each kernel, the optimization flags used to compile the kernel have to be specified.

Then the roofline controller is instructed to add a roofline point to the plot.

The first argument is the series name, next the label of the data point. Points with the same series name are connected with a line in the plot. The rest of the arguments specify the kernel and how the required quantities should be measured.

In the rest of the loop, the throughput and the operation count are measured and printed to the console. This is an example of how to use the quantity measuring service.

The last statement of the measure() body causes the plot to be output to a file in the current directory. This involves the invocation of gnuplot.

The addRooflinePoint() method of the **RooflineController** and the measureQuantities() method of the **QuantityMeasuringService** depend on a lot of functionality of the measurement driver and the measuring core. First a measurement was created from the kernel and the measurers required to measure the requested quantities. Then it was checked if there is already a result for the measurement in the cache. If not, the measurement was serialized, the measuring core was configured, built and started. Then the result of the core was parsed and stored in the cache. And finally, the requested quantities were calculated.

The only measurement specific part involved in this process is the implementation of the kernel. First the header:

```
class TriadKernel : public TriadKernelData{
    double *a,*b,*c;

protected:
    std::vector<std::pair<void*,long> > getBuffers();

public:
    void initialize();
    void run();
    void dispose();
};
```

All declared methods override methods from **KernelBase**. The kernel requires three buffers. The buffers are allocated and initialized in initialize() and freed in dispose(). getBuffers() returns the buffers along with their sizes. This is used to clear the or warm the caches. run() contains the actual algorithm.

```
void TriadKernel::initialize() {
    srand48(0);
    size_t size = getBufferSize() * sizeof(double);

    // allocate the buffers
    a = (double*) malloc(size);
    b = (double*) malloc(size);
    c = (double*) malloc(size);

    // initialize the buffers
    for (long i=0; i<getBufferSize(); i++){
        a[i]=drand48();
        b[i]=drand48();
        c[i]=drand48();
```

```cpp
    }
}

std::vector<std::pair<void*, long> > TriadKernel::
  getBuffers() {
    size_t size = getBufferSize() * sizeof(double);

    std::vector<std::pair<void*, long> > result;
    result.push_back(std::make_pair((void*) a, size));
    result.push_back(std::make_pair((void*) b, size));
    result.push_back(std::make_pair((void*) c, size));
    return result;
}

void TriadKernel::run() {
    for (long p = 0; p < 1; p++) {
        for (long i = 0; i < getBufferSize(); i++) {
            a[i] = b[i] + 2.34 * c[i];
        }
    }
}

void TriadKernel::dispose() {
    free(a);
    free(b);
    free(c);
}
```

A.3 Multi Language Infrastructure

The shared entities are used from both C++ and Java. To avoid having to manually synchronize two versions of the same class, the source code for the C++ and the Java implementation is generated from an XML definition by the Shared Entity Generator. The XML definition contains class and field definitions only, no code. If class specific code is needed, it has to be implemented separately for each language and merged with the field definitions using inheritance.

The shared entity definitions, written in XML, are parsed using a serialization library called XStream. XStream maps classes to an XML representation. The classes used to define the shared entities are shown in Figure A.1.

The following is an example of a class definition:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<class name="MultiLanguageTestClass"
  cBaseType="MultiLanguageObjectBase"
  javaBaseType=""
  comment="Multi Language Class used for unit tests">

  <field
```

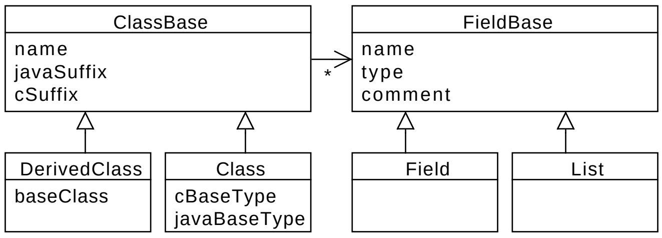


Figure A.1: Classes representing a multi language class definition

```
    name="longField"
    type="long"
    comment="test field with type 'long'"/>
  <list
    name="referenceList"
    type="MultiLanguageTestClass"
    comment="list referencing full classes"/>
  <field
    name="referenceField"
    type="MultiLanguageTestClass"
    comment="field referencing another class"/>
</class>
```

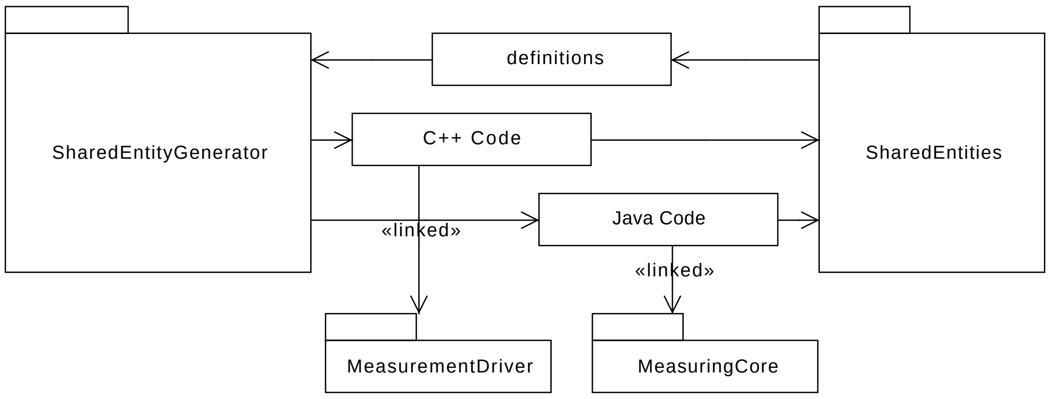
After the definitions are loaded, Velocity templates are used to generate all source code.

A normal entity has a C and a Java base type. The C base type has to directly or indirectly inherit from **SharedEntityBase**, which is a polymorphic class. This allows to use the RTTI (RunTime Type Information). Java base types have no such constraint (due to the implicit common base class Object). The base types are just included in the generated source code, but have no other effect on the code generation.

A derived entity names another entity as base type. The C and Java base types are set to that class. The fields of the base class are included in the serialization process. If just the C and Java base types would be set to the shared entity used as base class, the generated class would still derive from the base class, but the fields of the base class would not be included in the serialization process.

Often it is necessary to mix hand written code with the generated code. To support this, a suffix can be specified, which is added to the name of the generated class. Only the name of the generated class is affected, not the type name used for references to the class. A class named without the specified suffix has to be provided manually, and should derive from the generated class. Any additional code as well as additional fields can be included in the hand written class.

The class definitions and the generated code is located in the Multi Language Classes project. The generated Java code is linked by the Measurement Driver project. The generated C code is linked by the Measuring Core. The following Diagram shows these dependencies:



A.3.1 Serialization and Deserialization

Along with the source code for each class, a service serializing and deserializing multi language objects to/from a simple text based format is generated for both languages. It supports the following primitive types:

- double
- integer
- long integer
- boolean
- string

References to other shared entities are supported. The serializer can handle general object graphs.

Lists containing one of the supported primitive types as well as containing references to other shared entities are supported.

The service implementations for both languages follow the same structure. Each has two methods, one for serialization and one for deserialization.

The serialization method receives an object and an output stream. The method body contains an if for each known serializable class, which checks if the class of the object received is equal to the serializable class. If true, the value of all fields of the class and it's base classes get serialized. For references, the serialization method is called recursively with the same output stream and the referenced object as parameters.

The deserialization method works analogous to the serialization method. It receives an input stream. The method body contains an if for each known serializable class, which checks if the next line of the input names the serializable class. If true, a new instance of the class is created and the value of all fields are read from the input and set on the created instance, including all fields declared in a base class. If a reference is encountered, the deserialization method is called recursively with the same input, and the returned instance is used as field value.

A.4 Frontend

The measurement tool is a console tool controlled using command line options. Measurement results are either directly displayed on the console, dumped to a data file or processed, usually for generating a graph. The graph is typically stored as a file. But unlike normal tools, the source code is expected to change frequently, and the user likely switches often between coding and using the tool.

To support this usage pattern, the build process has been integrated into the normal tool operation. The frontend is used to first trigger the build process and then invoke the measurement driver. Otherwise, the user would have to keep two console windows open, one for building and one for measuring, and not to forget building to see the changes made to the source code.

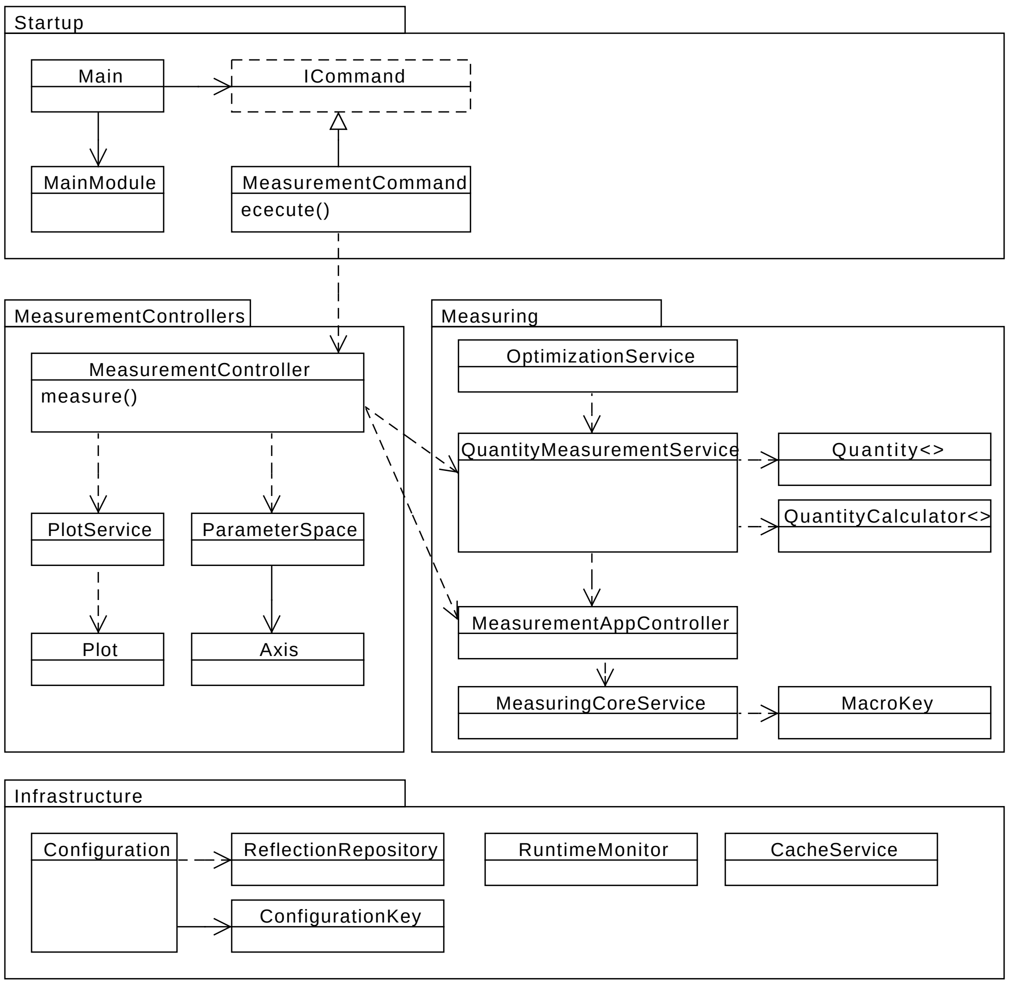
After building, the frontend starts the the measurement driver, forwarding it's own command line. Certain flags are used to control the operation of the fronted. These are not forwarded.

The frontend has a configuration system. The known configuration keys are defined at the top of the Main class. There are three configuration sources. The default configuration stored in a configuration file. It contains templates which are expanded during the build process. The result is included as resource in the generated JAR file. Flags of the default configuration can be overwritten using a user configuration file, which is located by default under " /.roofline/frontendconfig". This location can be changed using a command line argument. Finally, the command line options known by the frontend are used to modify the configuration flags after they have been loaded.

A.5 Measurement Driver

For the design of the measurement driver, we used software engineering best practice, namely unit testing (junit), mocks (jmock), and dependency injection (guice). Further a domain model (DOM), controllers, repositories and stateless services as described in [9]. Describing all these concepts lies beyond the scope of this report. It is assumed that the reader has a basic understanding of the mentioned concepts.

The following diagram shows an overview of the driver:



In the following paragraphs, we will have a quick look at the look at the different parts.

The entry point of the driver is the **Main** class. First, the dependency injection framework is initialized. This is accomplished using the **MainModule**. It's configure() method uses the **ClassFinder** to find all compiled classes and configures how they are instantiated, mainly based on naming conventions.

Then the command line arguments are parsed. If command line auto completion is desired (indicated by the '-autocomplete' flag), the auto completion process starts.

Otherwise the configuration is initialized, using the flags specified at the command line and the configuration files (default configuration stored in the jar and user configuration from the home directory of the user).

The last step in the initialization sequence is to set up log4j, the logging framework.

Then the class for the command given on the command line is instantiated and the execute() method on the resulting **ICommand** is called.

When a 'measure' command is given, a **MeasurementCommandController** is instantiated. The command controller looks a the next command line argument, instantiates the corresponding measurement controller and calls the measure() method.

The measurement controller will typically create multiple **Measurement**s with different parameters. The **ParameterSpace** facilitates iterating over all possible parameter combinations. Each parameter is associated to an **Axis**. For each axis, one or multiple values can be given. The getAllPoints() returns a **Coordinate** for each possible value combination. Itearating over the points in the space, the measurement controller can construct a measurement for each point.

The results of the constructed measurements can be either printed to the console, or stored in one of various **Plot**s. When all data is gathered, the plot can be rendered and written to an output file using the **PlotService**.

Although it is possible to directly create the **Measurer**s required to measure something, most of the time the intent is to measure a certain **Quantity** (**OperationCount**, **TransferredBytes**, **Performance** etc). The **QuantityMeasuringService** can be used to obtain a **QuantityCalculator** for a quantity. The calculator can be queried for the list of measurers which are required to calculate the quantity. When the results of all required measurers are known, they can be passed to the calculator, which will return the desired quantity. In addition, the quantity measuring service provides convenience methods for working with the quantity calculators.

Once the **Measurement** is constructed, the measure() method of the **MeasurementAppController** is used to perform the measurement. First it is checked if a cached result is available for the measurement (using the **CacheService**). If not, the **MeasuringCoreService** is used to build the core for the measurement and to start the core.

The **Kernel**s can define macros. During build preparation, all macro definitions present in the measurement are collected and written to generated header files within the core.

During the execution of the measurement driver, the run time used for the various tasks is collected using the **RuntimeMonitor**. At the end of the execution, the times spent for the tasks is printed to the console.

A.5.1 Dependency Injection Configuration

Generally, a convention over configuration approach was chosen for the configuration of the dependency injection. The conventions as well as optional exceptions are defined in the MainModule. The conventions are:

Services all classes in the services packages are bound to themselves as singletons

Repositories all classes in the repositories packages are bound to themselves as singletons

Application Controllers all classes in the 'appControllers' package are bound to themselves as singletons

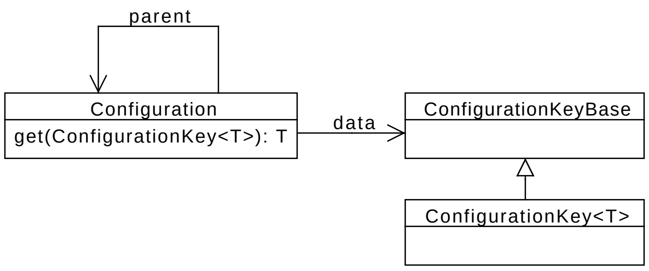
Measurement Controllers all classes deriving form IMeasurementController in the 'measurementControllers' package are bound to the IMeasurementController interface annotated with their name

Commands all classes deriving from ICommand in the 'commands' package are bound to the ICommand interface annotated with their name

A.5.2 Configuration

The design goal was to create a configuration system which

- allows to set configuration flags from the command line and from configuration files
- can manage some form of comment for the flags
- makes the available flags transparent
- supports user specific configurations
- allows the change of the configuration flags at runtime



The central class of our solution is the **ConfigurationKey**. A configuration key contains a string key which identifies the configuration flag it represents. In addition, it contains a description and the default value of the flag. It has a template parameter which defines the data type of the flag. This removes the necessity to use type casts when reading configuration flags. Configuration keys should be stored in public static variables. The help command scans all classes of the measurement driver for such configuration keys and prints the key string and the description. After the configuration is loaded, it is checked if a configuration key is defined for each configuration flag specified. If a configuration key for a flag is missing (or more likely, a configuration flag has been misspelled in the configuration) an error is generated.

The values associated with configuration keys are stored in **Configuration**s. Configurations can be chained together using the parent links. If no value is found in a configuration or all of its ancestors, the default value stored in the configuration key is used.

The state of a configuration can be saved on a stack using `push()` and restored using `pop()`. All modifications to a configuration after a push are undone by the pop. This can be used to temporarily change the configuration.

The following paragraphs describe the sources of configuration flag definitions in order of decreasing precedence.

The command line is scanned for arguments starting with a dash. Such arguments are expected to be in the form of ”-<flag key>=<value>” and specifies that the configuration with the specified flag key should have the specified value. Configuration flag definitions on the command line have highest precedence.

Next come two configuration files. They both have the same format: each line consists of the flag key, followed by an equal sign and the flag value.

The first file is the user configuration file. By default it is located under ˜/.roofline/config, but this can be changed using the ”userConfigFile” configuration flag, in particular by overwriting the flag on the command line.

The second file is the default configuration. It is located in the source code of the measurement driver, and can be loaded from the classpath. It contains some placeholders, which are expanded during the build process.

Finally, the flag definitions with lowest precedence are the default values given in the configuration keys.

A.5.3 Auto Completion

To enable auto completion support, a small shell script has to be placed in a system directory (see B.1). Whenever auto completion is requested by the user on the command line (by pressing TAB), to ‘rot’ tool is called with the ‘-autocomplete’ flag.

The frontend just forwards to the measurement driver, without recompilation. The driver then provides the completion proposals, taking registered commands and measurement controllers into account.

A.5.4 Commands

A command is represented by a class deriving from **ICommandController** and should be placed in the ‘commandControllers’ package. A command has a name and a description, which should be the return value of the getName() respectively getDescription() methods of the command. The measurement driver expects a command name as first argument. The name is matched against the names of all available commands. If a command matches, the execute() method of a new instance of the corresponding class is called with the remaining command line arguments as parameter.

A.5.5 Measurement Controllers

The operation of the measurement driver is controlled by the measurement controllers. They define which measurements to perform and how to process the output. The measure command instantiates a measurement controller and calls the measure() method.

A.5.6 Parameter Space

When implementing measurement controllers, one often has to iterate over all possible combinations of some parameters. The **ParameterSpace** was designed to support this.

Every parameter is identified by an **Axis**. For each axis, one or multiple values are specified. After the desired values are specified, all possible parameter combinations can be generated, represented by **Coordinate** objects. The points are generated implicitly when iterating over the space.

Example:

```
space.add(systemLoadAxis, SystemLoad.Idle);
space.add(systemLoadAxis, SystemLoad.DiskOther);
space.add(systemLoadAxis, SystemLoad.DiskAll);
space.add(systemLoadAxis, SystemLoad.AddOther);
space.add(systemLoadAxis, SystemLoad.AddAll);

space.add(clockTypeAxis, ClockType.CoreCycles);
space.add(clockTypeAxis, ClockType.ReferenceCycles);
space.add(clockTypeAxis, ClockType.uSecs);

for (Coordinate coordinate : space) {

    ClockType clockType
        = coordinate.get(clockTypeAxis);
    SystemLoad systemLoad
        = coordinate.get(systemLoadAxis);
    ...
}
```

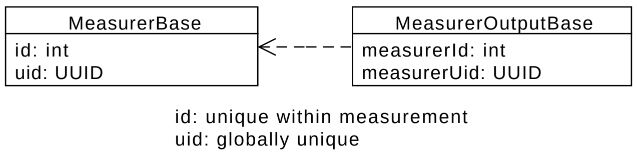
To facilitate the initialization of measurements, the classes of the measurement description have an initialize() method which takes a coordinate as parameter. Depending on the kernel or measurer at hand, some fields are set to the value of an axis given by the coordinate.

The most common axes are defined in the **Axes** class.

A.5.7 Retrieving Outputs

To process the results of a measurement, it is frequently necessary to retrieve the output of a specific measurer.

The straight forward approach would be to give each measurer an unique id, and to store the id of the measurer with the measurer output. Measurers are newly created with each invocation of the measurement driver, possibly leading to new ids. But for caching, the ids of the measurers do not matter.



To overcome these problems, two ids are generated for each measurer. One identifier uniquely identifying each instantiated measurer. And an id which is unique within one measurement. When loading a result from cache, the unique identifiers of the loaded result are set to the identifiers of the measurement at hand.

To retrieve the output of a measurer, the **MeasurementResult** and the **MeasurementRunOutput** provide several methods which take a measurer as argument and return it's output.

A.5.8 Plotting

Generating plots is an important feature of the tool. There are different plot types. The most important are roofline plots and distribution plots. A **RooflinePlot** contains all data required to generate a roofline plot, including peak performances and memory bandwidths.

A **DistributionPlot** is a 2D plot containing multiple data series. For every discrete x coordinate, there can be multiple values per series. The values are shown using an error box for the 25th/75th percentile and whisker bars for the minimum and maximum.

Once an instance of a **Plot** is filled with data, it can be passed to one of the plot() methods of the **PlotService**. A gnuplot script and the required data file will be created in the current directory. Then gnuplot is called to generate a PDF of the plot.

A.5.9 The MeasurementAppController

The measurement application controller is the entry point for performing measurements. It is the sole client to the **MeasuringCoreService**, which provides the low level control over the measuring core, and keeps track of the measurement the core is compiled for.

In addition, it uses the **MeasurementHashRepository** to keep track of measurements which have equal measuring cores.

The main method of the controller is measure(). Functionality in pseudo Code:

```
MeasurementResult measure(measurement, numberOfRuns)
"prepare measurement"
runOutputs=[]
if (useCachedResult || "measurement has been seen")
    loaded="load stored results"
    if (loaded!=null)
        if (!shouldCheckCoreHash
            || currentCoreHash==loaded.coreHash)
            runOutputs=loaded

if ("more results needed")
    newResult=performMeasurement()
    "merge and store loaded and new run outputs"

"build and return MeasurementResult with the desired
   number of run outputs"
```

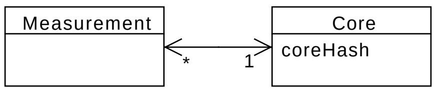
The method first tries to load a measurement result from the result cache. If not enough run outputs are loaded (or none at all), the measurement is performed to get the remaining measurement run outputs.

Finally, a measurement result with exactly the requested number of runs is constructed and returned.

It is possible to disable loading stored results by setting the useCachedResults configuration key to false. The measurement is performed and existing results are overwritten.

The hash code of the core is stored along with the measurement result. This allows to check if the currently compiled core is equal to the core which was used during the measurement. By default, if the core changed since the results were generated, the results are not used and new results are generated using the current measuring core. By setting shouldCheckCoreHash to false, this check can be skipped.

Preparing and building the measuring core are expensive operations in term of runtime. Therefore, the measurement application controller keeps track of as much information about the measurements and the cores needed to perform them as possible. The **MeasurementHashRepository** is used for this purpose. It has the following internal Model:



The **Core** class is private and does not leave the repository.

The model is exposed through the following methods

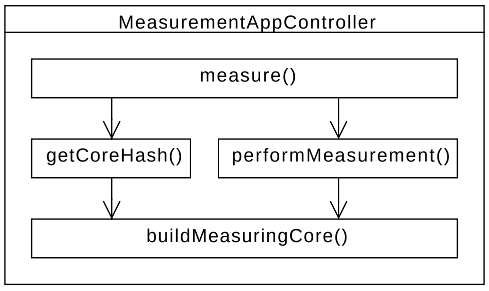
- areCoresEqual(measurementA, measurementB): bool
- setHaveEqualCores(measurementA, measurementB)
- setCoreHash(measurement,coreHash)
- getCoreHash(measurement): CoreHash

Before building, the controller asks the repository if the core for the new measurement is the same as the currently built one. (using areCoresEqual()) If the cores are the same, no building is required.

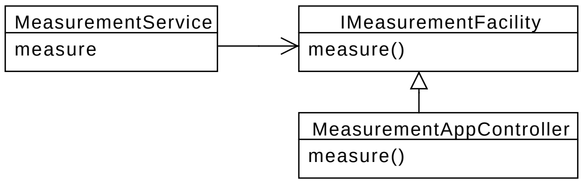
During the build preparation the controller and the **MeasuringCoreService** monitor changes to the core. If no changes were necessary, the repository is notified using setHaveEqualCores(). The cores of the two measurements are merged.

When a core hash is required (for example to check if the current core is the same as the one used to generate a stored result), the controller first asks the repository for it using getCoreHash(). If the hash is not known, the core is built for the measurement, the hash is calculated from the core and stored in the repository using setCoreHash(). This could again lead to a merging of two cores, if a core with the same hash is present already.

Building the measuring core can become necessary when the core hash of a measurement has to be known, or when a measurement is to be performed. This is reflected in the call graph of the methods within the application controller:



Since it makes sense that services can start measurements, the **MeasurementService** provides a measure() method. The service knows an instance of **IMeasurementFacility** which provides measure(), and forwards all calls to its own measure() method to the measurement facility. The facility is is implemented by the **MeasurementAppController** controller. Therefore the service ultimately forwards all calls to measure() to the application controller.



A.5.10 Architecture Specific Behavior

The measurement driver supports multiple system architectures. Currently, the system architecture is identified by the available PMUs. When a performance event is to be read, a list with the event for each architecture is passed to the getAvailableEvent() method of the **SystemInfoService**. The method returns the available event. In other cases, the presence of a PMU is checked directly.

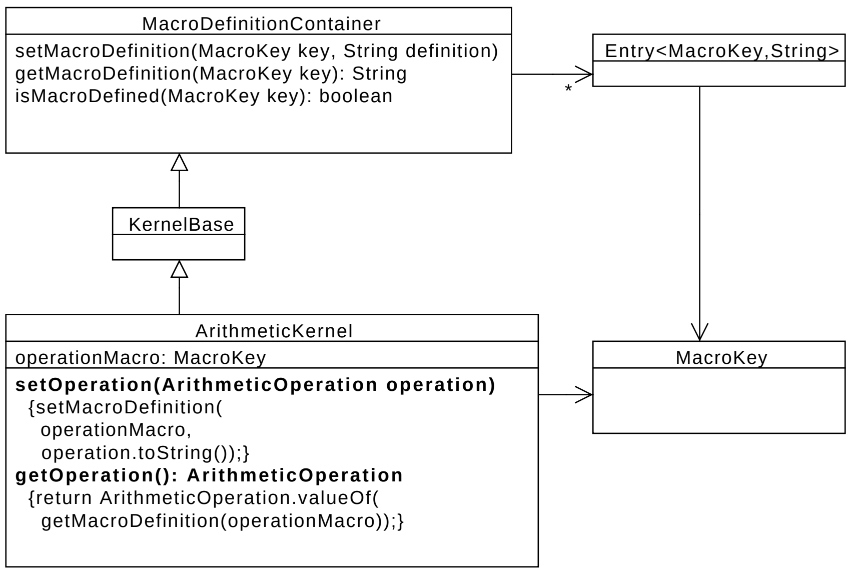
For further development, it might become beneficial to use the specification pattern.

A.5.11 Preprocessor Macros

Preprocessor macros are used to allow flexible compile time parameterization of the measuring core. The macros are defined by the measurement driver. Before the compilation of the measuring core, the measurement driver writes the definition of each macro to a separate include file. This allows the build system to track macro definition changes for and to recompile only the required parts.

In the measurement driver, each macro is identified by a macro key, which contains the macro name, a description and the default value. The macro definitions are stored in classes deriving from MacroDefinitionContainer. The classes should define macro keys by placing them in private static variables. To access the macro definition, getters and setters have to be provided.

When the measuring core is configured to perform a measurement, the macro keys are collected from the classes of the measurement driver using reflection. Then the macro definitions are extracted from the measurement definition and referenced objects. If no definition is given for a macro, the default definition found in the macro key is used. If contradicting definitions are found, an error is raised.


MacroDefinitionContainer
setMacroDefinition(MacroKey key, String definition)
getMacroDefinition(MacroKey key): String
isMacroDefined(MacroKey key): boolean
Entry<MacroKey,String>
*
KernelBase
ArithmeticKernel
operationMacro: MacroKey
setOperation(ArithmeticOperation operation)
{setMacroDefinition(
operationMacro,
operation.toString());}
getOperation(): ArithmeticOperation
{return ArithmeticOperation.valueOf(
getMacroDefinition(operationMacro));}
MacroKey

A.5.12 Measurement Result Caching

The measurement controllers mix the definition of the measurement parameters and the processing of the output. Thus, if the output processing logic needs to be modified, the measurements have to be performed again. This causes a delay, which is avoided by caching the measurement results.

All parameters of a measurement are contained within the measurement description and the referenced objects. Therefore, if the measurement description is identical to a measurement description of a previous measurement, the result of the previous measurement can be reused.

The cache mechanism works using a hash function on the XML representation of the measurement description. After a measurement has been performed, a file named after the hash value of the measurement description of the measurement is created and the measurement results are stored therein. Before a measurement is performed, the hash value is computed. If a corresponding file is found, the previous measurement results are reused.

A.6 Measuring Core

The measuring core is based on the object graph constructed by the driver. The classes are extended with code and data.

A.6.1 Core Architecture

To fully utilize multi core systems, applications have to be implemented with multiple threads or processes. Measuring such applications is considerably more difficult than measuring single threaded applications. If the thread management is implemented specifically for the measurement at hand, the measuring code can be weaved into it by hand. But if the threads or processes are created within legacy or closed source code, the measuring tool has to take care of detecting the creation of threads and install the necessary measurers. For the roofline measuring tool we will only consider multi threaded applications.

To gain full control over the kernel code, the measurement tool starts the kernel within a child process. The parent process attaches to the child using ptrace. This causes the child to be stopped when certain events occur and the parent is notified. The events include thread creation and breakpoints.

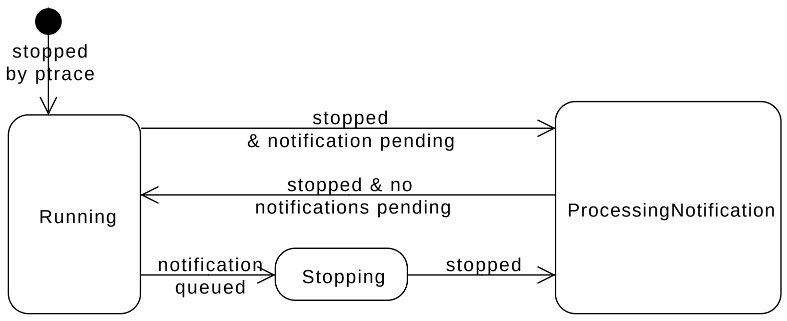
Every kernel thread can raise events at any point. The events include thread creation, breakpoints, starting and stopping of workloads etc. Whenever an event occurs, the rule list has to be searched and the matching actions have to be executed. The rule list is always searched in the thread which raised the event. If the event caused the thread to be stopped and the parent thread to be notified, the parent restarts the thread with a notification of the event which occurred. The thread will search the rule list and continue execution.

It is important to distinguish the events from notifications in this context. An event is handled by the rule list of the child and typically generated by the child process as well. A notification is used to communicate between the child process and the parent process. However, the parent can notify (using a notification) the child of a certain observation, which causes the child to generate an event.

The rule list is always searched in the thread generating the event. If an action has to be executed in another thread, it has to be queued using ChildThread::queueAction()

The technically most challenging problem is to interrupt another thread and make it execute some action. There are two approaches to this problem, either using a signal handler of the thread or using ptrace to call code within the thread. Since installing a signal handler in a thread could cause unwanted side effects, we use the ptrace approach. SIGTRAP is sent to the target thread, which will cause it to be stopped. The parent modifies the thread state of the stopped thread. When the thread resumes execution, it executes some event handling code and then return to the location the thread was interrupted.

A.6.2 Child Thread States



The parent process manages a state for each child thread. When a child thread starts, ptrace will immediately stop it with SIGSTOP. If the notification system is ready, a ThreadStarted notification will be sent to the child.

While a notification is beeing processed by the child, it's state is set to processing. When no more notifications are pending, the state is set to running. If a notification is queued from one thread to another and the state of the receiving thread is running, a SIGTRAP is sent to the receiving thread and the state of the receiving thread is set to stopping. The stopping state indicates that the thread will stop eventually.

A thread can exit at any time. When this is detected by the parent process, the state and the notification queue are erased.

A.7 Thread Representation in the Child Process

In the child process, threads are represented as **ChildThread** objects. Since every **Workload** runs in it's own thread, a child thread is associated with every workload.

When a new thread is spawned, it will be stopped by ptrace. The parent sends the ThreadStarted notification to the child. In the handler, the child will instantiate the **ChildThread**. If the started thread is a workload thread, the startup routine of the workload will associate the instantiated child thread with the workload.

Actions can be sent from one thread to another using `ChildThread::queueAction()`.

A.7.1 Building

Each measurement can be performed with different compiler optimization flags and macro definitions. Therefore, the measuring core has to be rebuilt for each measurement, which makes rebuilding the measuring core a frequent operation. It should therefore be as fast as possible. This is achieved by carefully tracking all build dependencies and by using ccache.

CCache is a compiler cache. Whenever the compiler is run, ccache hashes all input files, together with the compiler flags. It then checks if it's cache already contains an entry for the hash value. If this is not the case, ccache runs the compiler and stores the output together with the hash value of the input in it's cache. If the hash value is present already, it does not run the compiler but uses the compiler output stored in it's cache. This considerably speeds up recompilations.

But ccache still has to build the hash values and copy the compiler output, which takes some time. This is where tracking the build dependencies comes in. The following parameters can change between measurements:

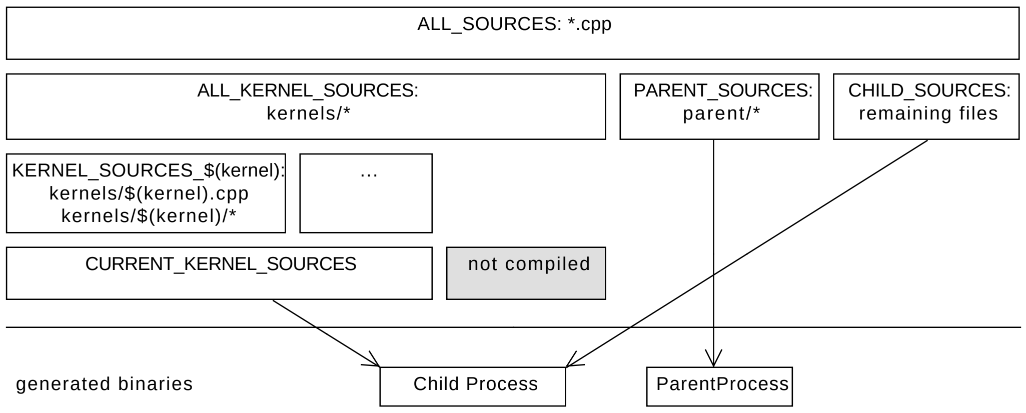
Macro definitions Each macro definition is stored in a separate file, which is only updated by the measurement driver if the macro definition changes. Every source file which needs a macro definition includes the corresponding file. These inclusions are tracked and allow to only recompile the affected source files.

Compiler flags Each kernel specifies it's own optimization flags. The compiler flags used for the rest of the measuring core do not affect the measurement results. The compiler flags are stored in a separate file. Whenever it is changed, the parts affected are recompiled.

Compiled Kernels For each measurement, only the kernels actually used are compiled. The measurement driver writes the kernel names into a separate file. The child binary is recompiled when it changes.

The build process is controlled using the gnu make utility [14]. Make automatically determines which parts of a program have to be recompiled, based on rules stored in a makefile. Each rule consists of target files, prerequisite files and a recipe. Make checks the modification times of the target and the prerequisite files. If any prerequisite is newer than any target, the recipe is executed in order to update the target files. The recipe is a sequence of shell commands.

The following diagram shows how the source files are categorized and compiled:


ALL_SOURCES: *.cpp
ALL_KERNEL_SOURCES: kernels/*
PARENT_SOURCES: parent/*
CHILD_SOURCES: remaining files
KERNEL_SOURCES_$(kernel): kernels/$(kernel).cpp kernels/$(kernel)/*
...
CURRENT_KERNEL_SOURCES
not compiled
generated binaries
Child Process
ParentProcess

The makefile used for the measuring core first instructs make to use the find utility to get a list of all source files (with .cpp extension) in the measuringCore/src and measuringCore/generated directories (ALL_SOURCES).

Using the filter functions of make, the kernel sources are set to the subset of ALL_SOURCES which is located in src/kernels or generated/kernels. These are all sources related to kernels. (ALL_KERNEL_SOURCES).

The sources of the parent process are the subset of ALL_SOURCES which is located in src/parent(PARENT_SOURCES).

The source files of ALL_SOURCES not contained in ALL_KERNEL_SOURCES or PARENT_SOURCES are stored in CHILD_SOURCES.

The names of all present kernels are stored by the measurement driver in generated/kernelNames.mk. For each of the kernels named there, the source file named after the kernel and all source files in the subdirectory named after the kernel collected in a variable (KERNEL_SOURCES_$(kernel)). They are compiled with the optimization flags of the compiler, wich are stored under generated/kernelOptimization.

The sources of all current sources are collected and form, together with the CHILD_SOURCES, the sources of the child process.

The parent process is compiled from the PARENT_SOURCES.

There is a rule without recipe with the kernel objects as target and the file containing the measurement specific optimization flags as prerequisite. This causes the file containing the optimization flags to be added as prerequisite for each kernel object file.

In the C programming language, it is possible to include other files in a source file. Of course, the compiled code depends on the contents of the included files, too. To track these build dependencies, the compiler is instructed to generate rules without recipes with the object file as target and the source file together with the included files as prerequisites. The generated rules are stored in .d files in the build directory and are included in the makefile.

If special compilation flags are required for a source file, a rule should be added near the end of the makefile.

A.7.2 System Initialization

We chose a modular approach to initialize the measuring core. Whenever a system part needs to run code when the program starts or shuts down, it can instantiate a class derived from SystemInitializer.

This is preferably achieved by declaring a global static variable named dummy in a .cpp file. Example:

```
// define and register a system initializer.
static class FooInitializer: public SystemInitializer{
  void start(){
    // code to be executed on startup
  }

  void stop(){
    // code to be executed on shutdown
  }
} dummy;
```

It is important to give every initializer subclass an individual name. Use the name of the file the initializer is declared in as prefix. If two initializer classes have the same name, they do not work correctly (instances of the wrong classes are created)

Whenever a **SystemInitializer** is instantiated, the instance is registered in a static global list. On system startup and shutdown, the start() respective stop() method of all registered **SystemInitializers** is called.

Appendix B

How To

B.1 Installation

see INSTALL file in tool directory

B.2 Create a New Kernel

First, you have to create an XML description of the new kernel in sharedEntities/definitions/kernels. Look at some of the other files in that directory and choose one as starting point for your own. Copy the chosen file and give it the name of your kernel. It must end with 'Kernel'. The file name is taken as class name for your kernel description.

Next, run 'rot help' to generate the source code from your XML description. In case your class defines a 'javaSuffix', the compilation following the source generation will fail. Create the Java code of your kernel class in measurementDriver/src/ch/ethz/ruediste/roofline/sharedEntities/Kernel and make it inherit from the generated class with the suffix. (see other kernels for examples)

The Java part of your kernel is now ready. You can use it in a measurement. But we are still missing the implementation in the measuring core. To implement the kernel, it is indispensable to specify a 'cSuffix' in your kernel description.

Create a class in measuringCore/src/sharedEntities/kernels, include the generated header file (filename contains the suffix) and derive from the generated class. Add fields for all the data buffers you plan to use (if any). Override and implement the following methods:

initialize() Allocate and initialize the required buffers.

getBuffers() Return the list of all buffers along with their size. This is used to automatically clear or warm the caches.

run() Run the kernel.

dispose() Free all buffers

In case you have additional code, you can place it in a subdirectory with the same name as your kernel, without the 'Kernel' suffix. All '.cpp' files will be compiled and linked with the measuring core.

B.3 Use the Driver as a Library

You can use the measurement driver as a library. First, run './gradlew –daemon measurementDriver:runnableJar' to create a jar containing the driver together with all dependencies. It will be located in 'measurementDriver/build/distributions/measurementDriver.jar'. Include the jar in your project.

During the startup of your application, call LibraryMain.initialize(). Now you can use the driver. You can directly instantiate entities and use Instantiator.instance.getInstance() to retrieve service instances.

B.4 Create New Measurement

- create new class in measurementDriver/measurements
- implement IMeasurement

B.5 Add Configuration Key

The configuration is used to set various flags in the measurement driver.

- add public static field of type **ConfigurationKey** to any class within the measurement driver.

B.6 Generate Annotated Assembly

- in Eclipse, hit build (Ctrl+B)
- change the kernel header file (make it recompile)
- build again
- from the console window, copy the compiler invocation for MeasurementSchemeRegistration.cpp
- open a terminal and go to tool/measuringCore/Debug.
- paste the compiler invocation
- insert "-Wa,-ahl=ass.s", check optimization flags
- issue command
- the annotated assembly code can be found in tool/measuringCore/Debug/ass.s
- open the annotated assembly code in Eclipse

Appendix C

Results on the Intel Core

We repeated our measurements on a desktop computer with an Intel Core 2 CPU. The results are not discussed in detail, but we tried to point out the most important differences to the results we measured for the Yonah architecture based system.

The CPU is an 'Intel Core 2 6600' (Family 6, Model 15, Stepping 6) running at 2.4 GHz. It contains two cores, each having a 32KB instruction and a 32KB data L1 cache, 8 ways set associative, with 64 bytes line size. The two cores share a 4MB unified L2 cache, 16 ways set associative and with 64 bytes line size. The TLB has 256 entries, 4 way set associative. The core frequency can scale between 1.6GHz and 2.4GHz. The bus frequency is 266MHz.

The main memory consists of two 1GB and two 512MB DDR2 modules, totaling in 3GB available memory. The theoretical throughput of the memory is 8.3 GB/s, which is 3.46 Bytes per core cycle, if the CPU runs at 2.4GHz.

C.1 Performance Counters

We used 'coreduo::UNHALTED_CORE_CYCLES' for measuring time. For the operation count, we used the follwing definitions for operation:

SinglePrecisionFlop SSE single precision operations.
'core::SSE_COMP_INSTRUCTIONS_RETIRED:SCALAR_SINGLE'
+2*'core::SSE_COMP_INSTRUCTIONS_RETIRED:PACKED_SINGLE'

DoublePrecisionFlop SSE double precision operations.
'core::SSE_COMP_INSTRUCTIONS_RETIRED:SCALAR_DOUBLE'
+2*'core::SSE_COMP_INSTRUCTIONS_RETIRED:PACKED_DOUBLE'

CompInstr Computational instructions retired. Counts SSE instructions and x87 instructions. Used for x87 code.
'coreduo::FP_COMP_OPSEXE'

SSEFlop SSE operations, sum of SinglePrecisionFlop and DoublePrecisionFlop

We used two variants of measuring the memory transfer volume. The **MemBus** variant uses 64*'core::BUS_TRANS_MEM', which measures the transfers on the system bus. The **MemL2** variant uses the counters for the L2 cache line allocation and eviction, namely 64*('core::L2_LINES_IN:SELF'+'core::L2_M_LINES_OUT:SELF'), combined with 8*‘core::SSE_PRE_EXEC:STORES’ to take non temporal stores into account.

We used XUbuntu 11.10, running a Linux 3.0.0-16 kernel in 64 bit mode and GCC 4.6.1.

C.2 Performance and Bandwidth Ceilings

For the Core architecture, one double precision addition and multiplication can be issued every cycle. The latencies are 3 cycles for the addition and 5 cycles for the multiplication. [12]

The throughput and latencies of the packed SSE instructions are equal to those of the x87 instructions. Thus with the x87 instruction set the theoretical ceiling is 1 flop/cycle for pure additions and multiplications. With the SSE instruction set the ceiling is 2 flop/cycle.

It should be possible to use the multiplier and the adder at the same time, so if there is an equal number of additions and multiplications it should be possible to reach 4 flop/cycle.

We used the same micro benchmarks for addition and multiplication as for the Yonah architecture.

For the mixed instruction set a different workload is used. Due to the latencies we always have to keep 5 multiplications and 3 additions in flight. This is achieved by the following code:

```
#define MULTIPLICATIONS 5
#define ADDITIONS 3
double mulR[MULTIPLICATIONS];
double addR[ADDITIONS];
...
#define MUL for (int h = 0; h < MULTIPLICATIONS; h++)\
                mulR[h] *= base;
#define ADD for (int h = 0; h < ADDITIONS; h++)\
                addR[h] += 1;
for (long i = 0; i < iterations; i++) {
    for (int p = 0; p < UNROLL; p++) {
        MUL ADD ADD MUL ADD ADD MUL ADD
    }
}
```

Optimizing the value for UNROLL and DLP in the range of 1 to 19 we found the following results:

Operation	Instruction Set	DLP	UNROLL	Performance [Flop/Cycle]	
				Measured	Theoretical
ADD	x87	9	15	0.998	1.0
	SSEScalar	9	15	0.998	1.0
	SSE	9	17	1.996	2.0
MUL	x87	10	15	1.998	1.0
	SSEScalar	10	15	0.998	1.0
	SSE	9	15	1.996	2.0
Mixed	x87	-	10	1.996	2.0
	SSEScalar	-	9	1.995	2.0
	SSE	-	11	3.991	4.0

In all cases we almost reach the theoretical values.

In our plots we show 2 flop per cycle as balanced scalar ceiling, 4 flops per cycle as balanced SSE ceiling and 8 flops per cycle as dual core SSE ceiling.

The bandwidth ceiling can be calculated from the parameters of the memory subsystem and is 3.46 bytes per cycle (see section C).

We use the same micro benchmarks as for the Yonah architecture. Optimizing loop unrolling and stream count resulted in the following numbers:

Operation	Streams	UNROLL	Throughput [Byte/Cycle]
Load	1	1	2.58
Write	1	3	1.63
Non Temporal	1	4	2.02

The results reported here were obtained using MemBus, but the results of the MemL2 variant are almost identical.

In the stream benchmark [5] the system reaches a copy rate of 3177 MB/s, which is $2 * 3434MB/1.8GHz = 2.84$ bytes per cycle. Our micro benchmark reported a smaller number, but the result is comparable to ours.

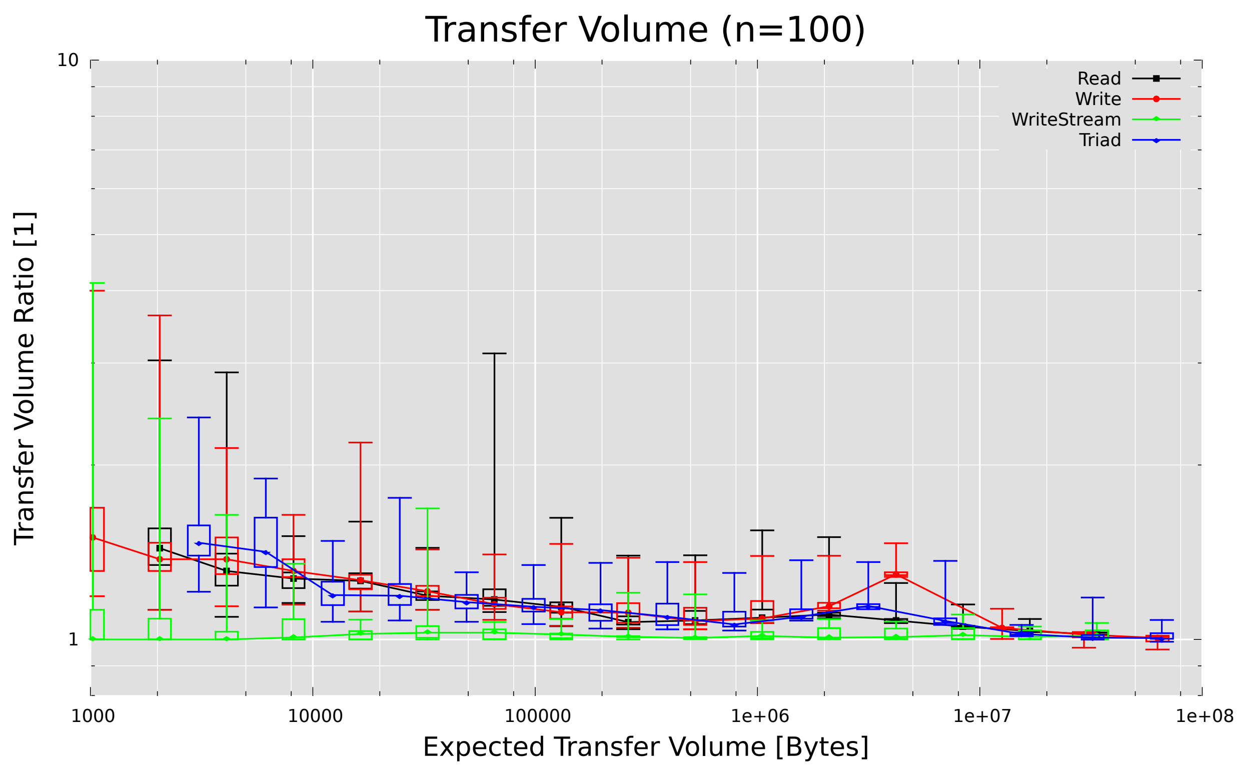
We also measure the random access throughput. A large buffer is allocated and randomly accessed. The throughput is 0.455 bytes per cycle. Since a cache line is 64 bytes, this results in one memory access every $64/0.455 = 141$ cycles.

In our roofline plots we only show the load ceiling (2.2 flops per cycle) and the random access ceiling (0.45 bytes per cycle).

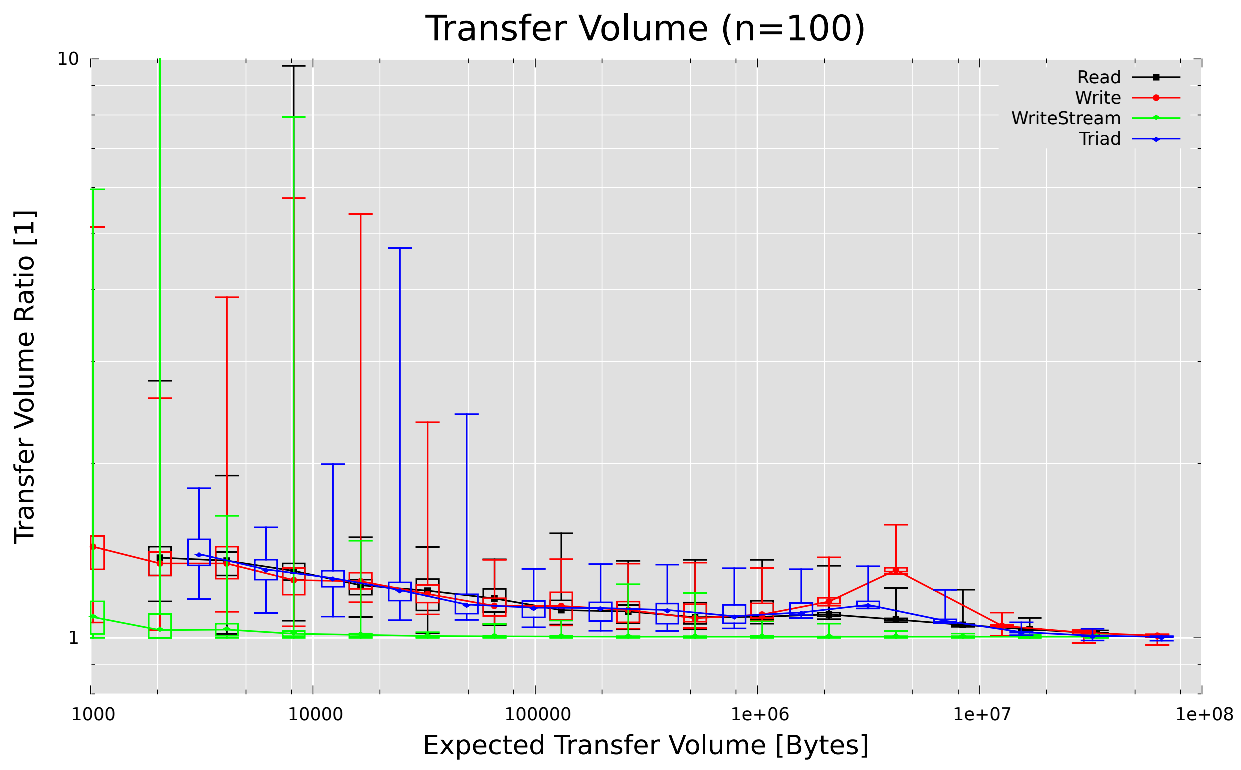
C.3 Measurement and Validation

The accuracy and precision are very similar to those of the Yonah architecture based system.

The most notable difference was observed for the measurement results of the transfer volume for multi threaded workloads when using MemBus. For the Yonah architecture, half of the measurement results showed the transfer volume of both threads and the other half only the volume of a single thread. On the Core only the volume of a single thread is observed.

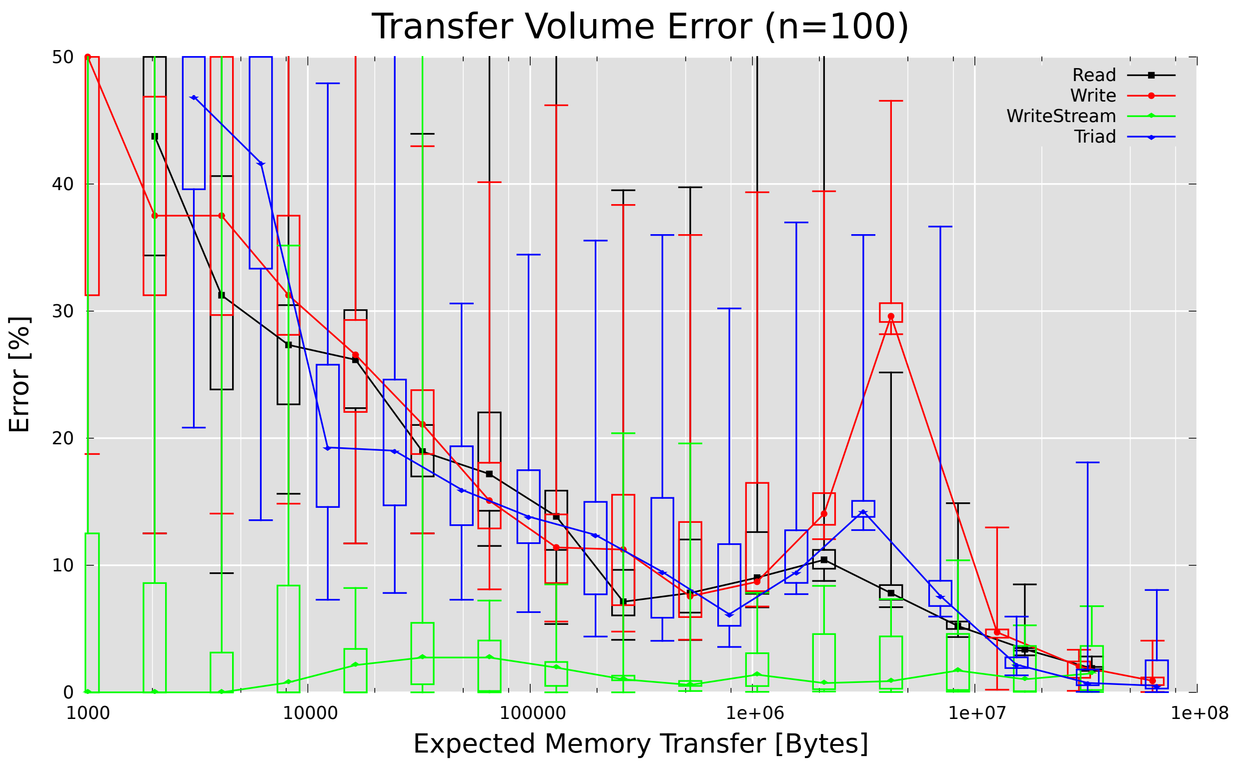


(a) MemBus

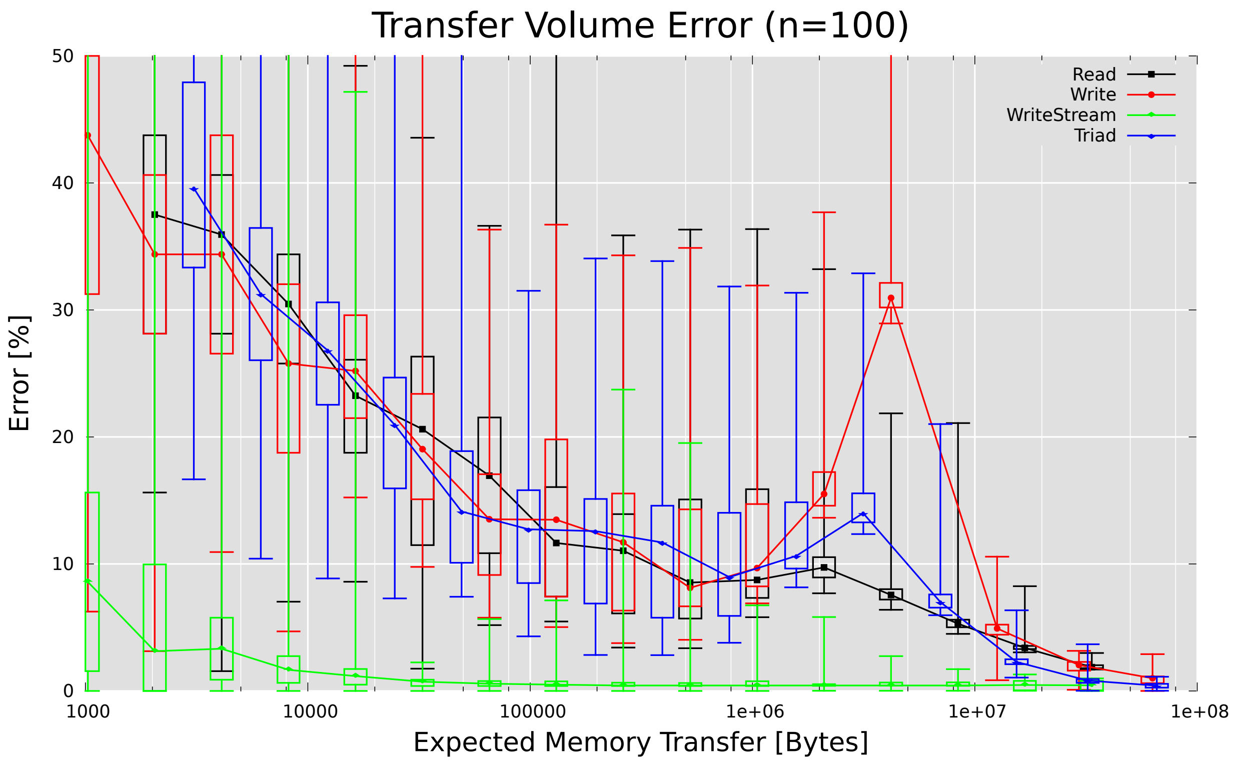


(b) MemL2

Figure C.1: Core: Ratio of the Actual Transfer Volume to the Expected Transfer Volume



(a) MemBus



(b) MemL2

Figure C.2: Core: Error of the Transfer Volume: $\mathrm{err}(\frac{\tau}{\Theta})$

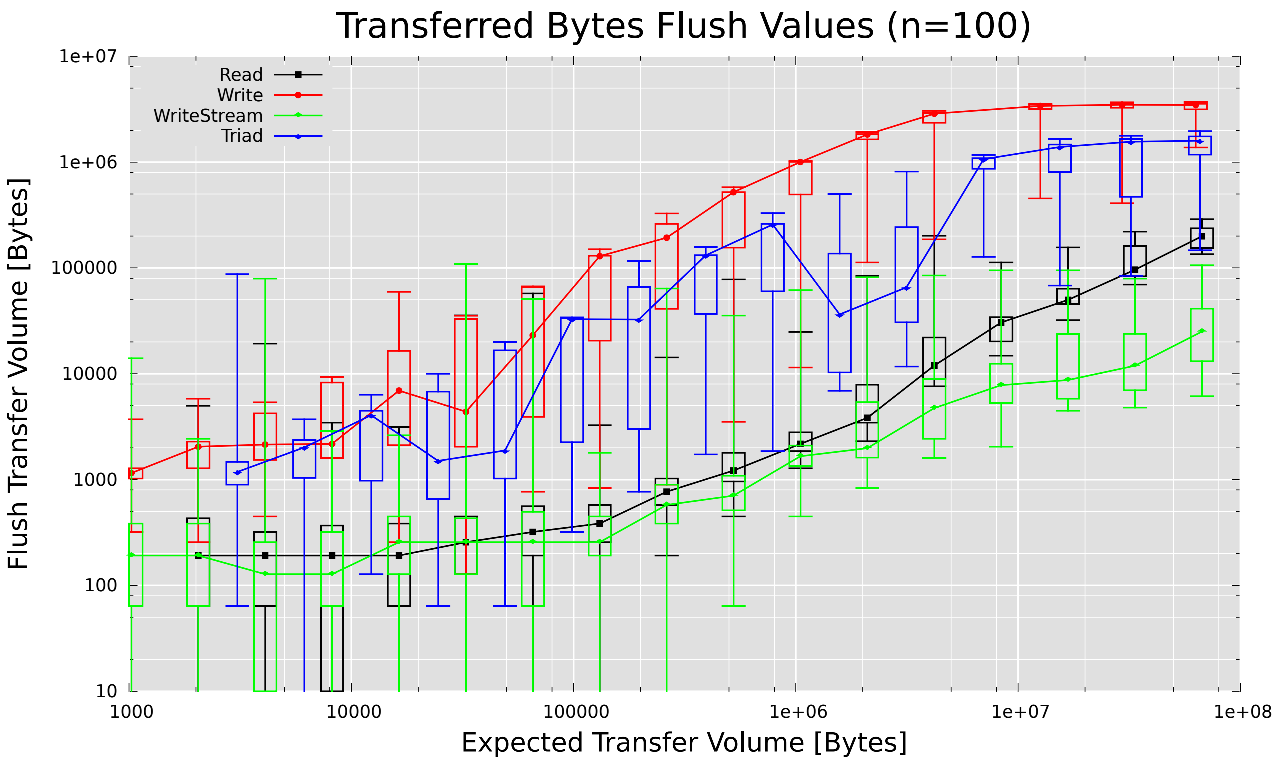
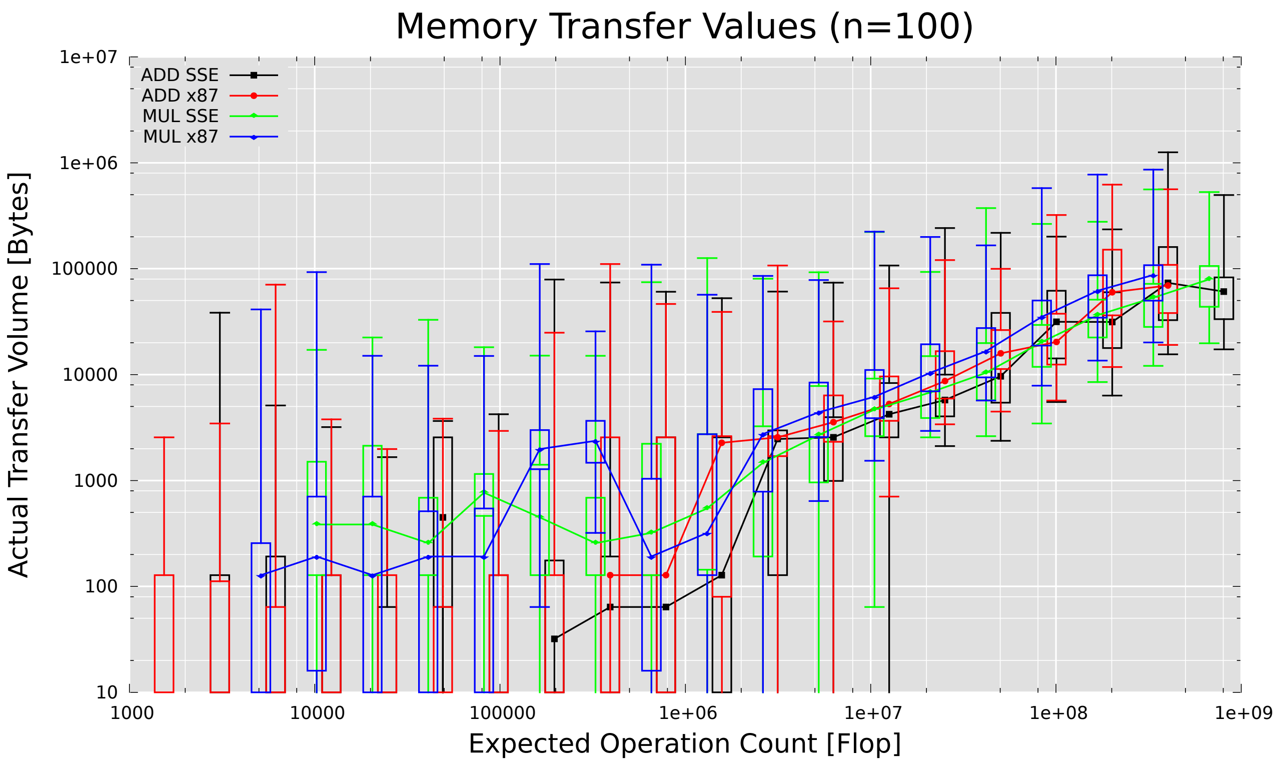
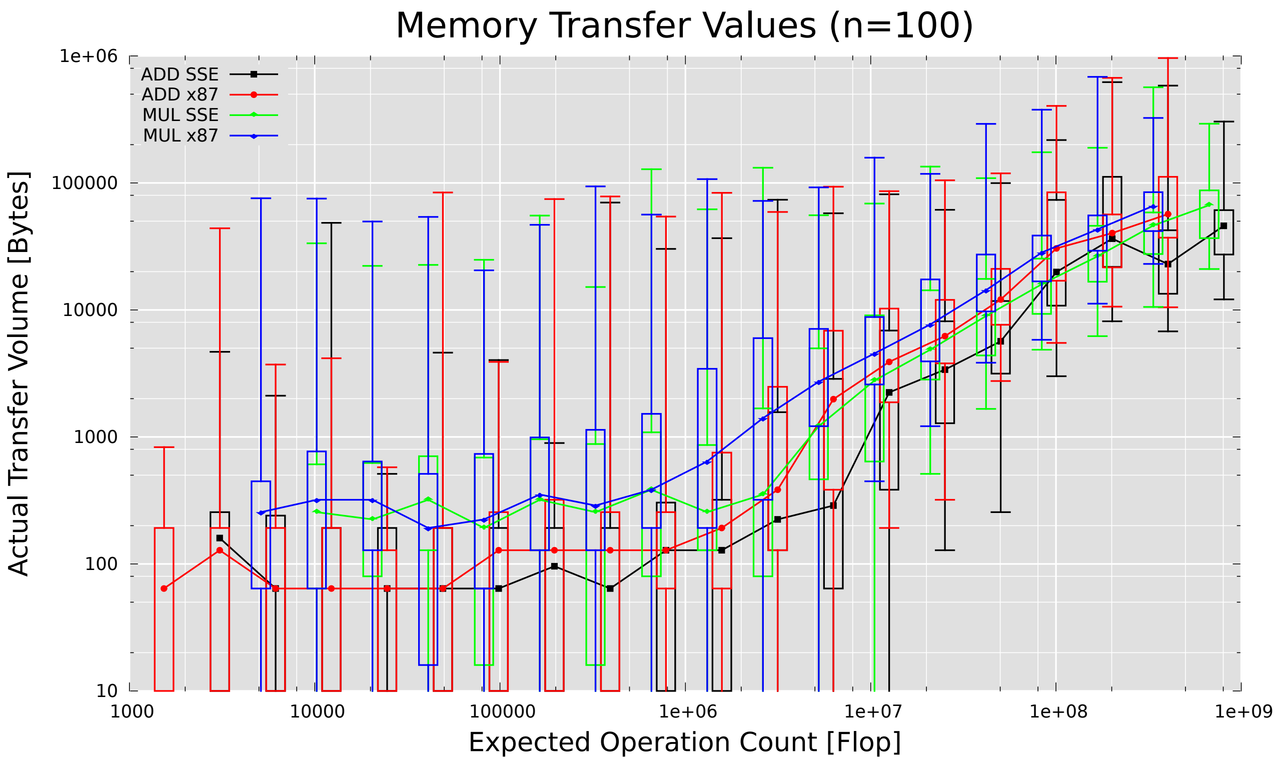


Figure C.3: Core: MemL2: Memory Transfer during Cache Flush after kernel execution



(a) MemBus



(b) MemL2

Figure C.4: Core: Transferred Bytes of the Arithmetic Kernels

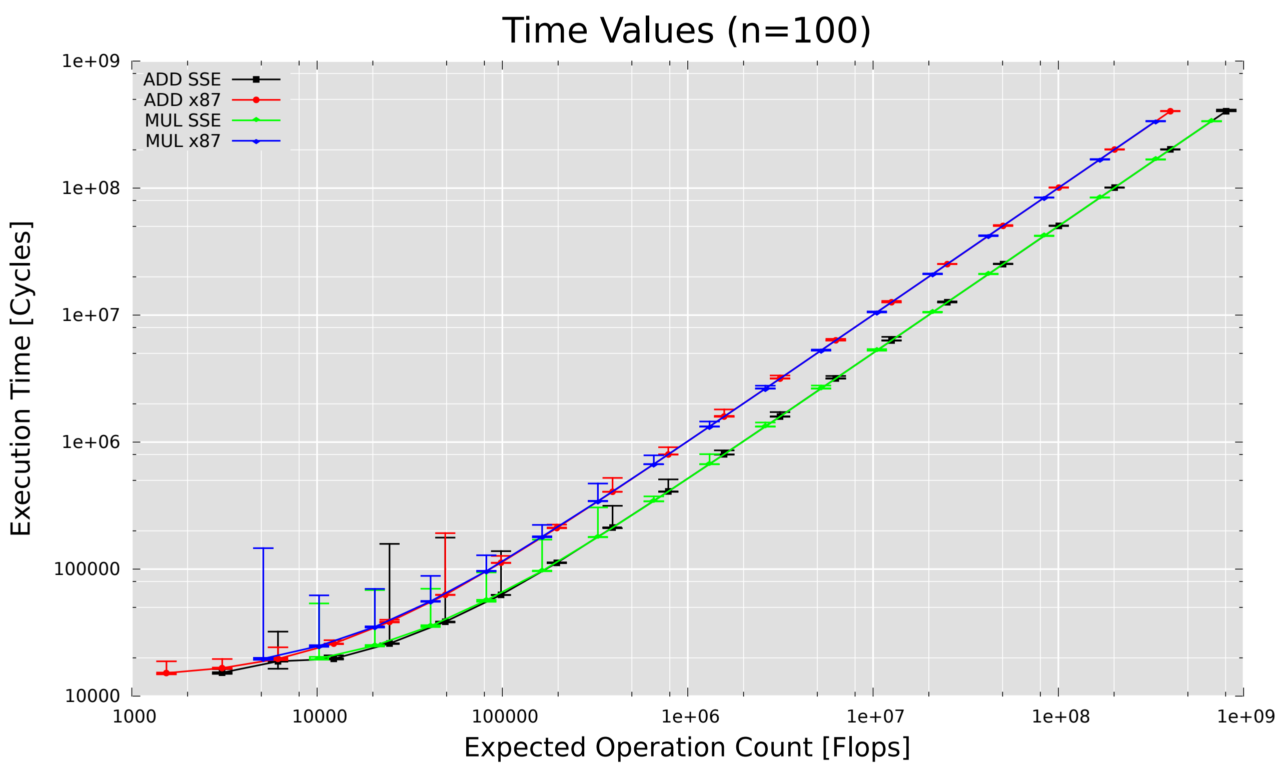


Figure C.5: Core: Execution Time of the Arithmetic Kernels

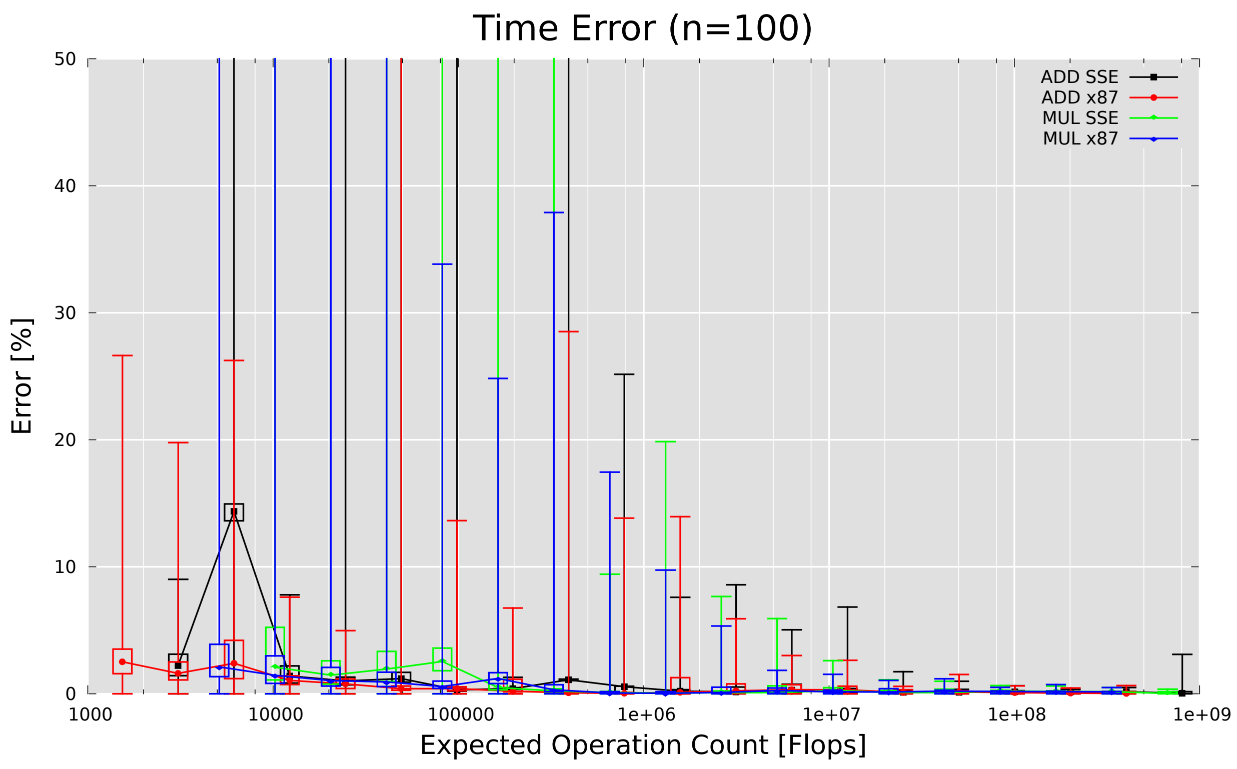


Figure C.6: Core: Error of the Execution Time of the Arithmetic Kernels: $\text{err}\left(\frac{\text{Execution Time}}{\min(\text{Execution Time})}\right)$

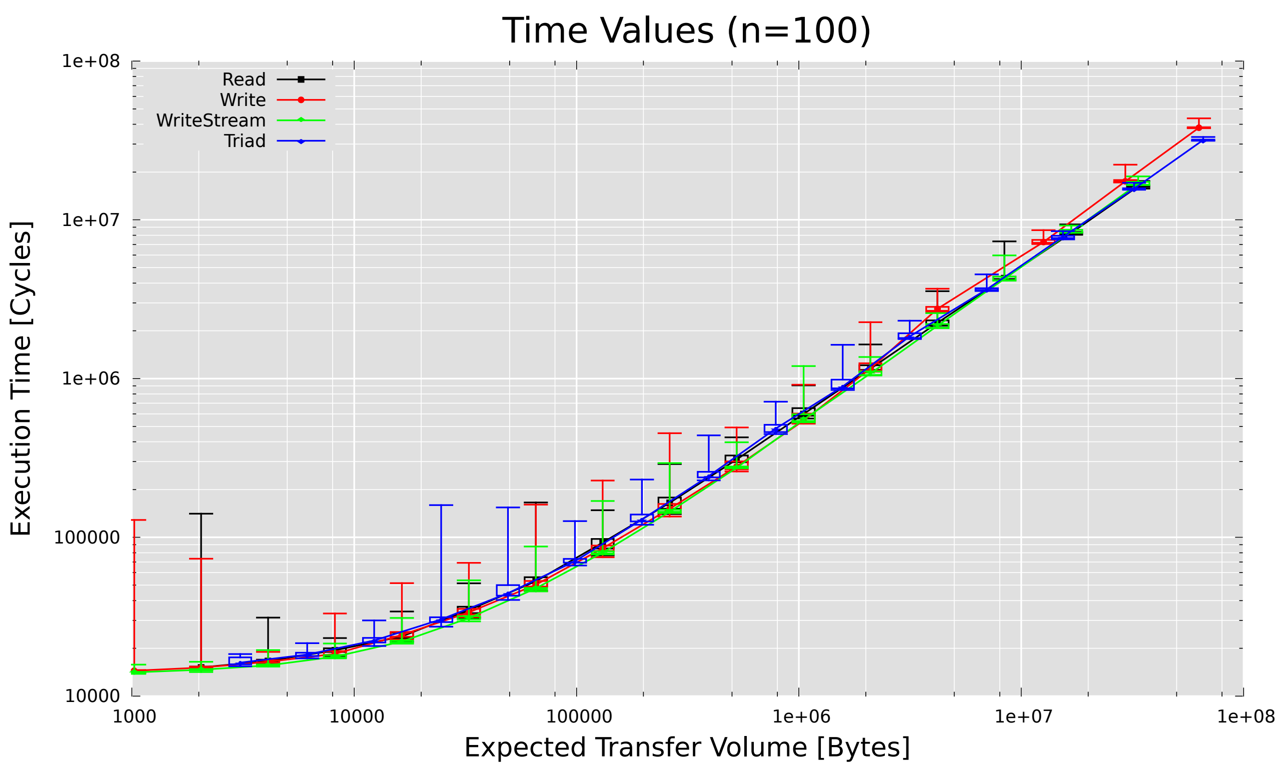


Figure C.7: Core: Execution Time of the memory kernels

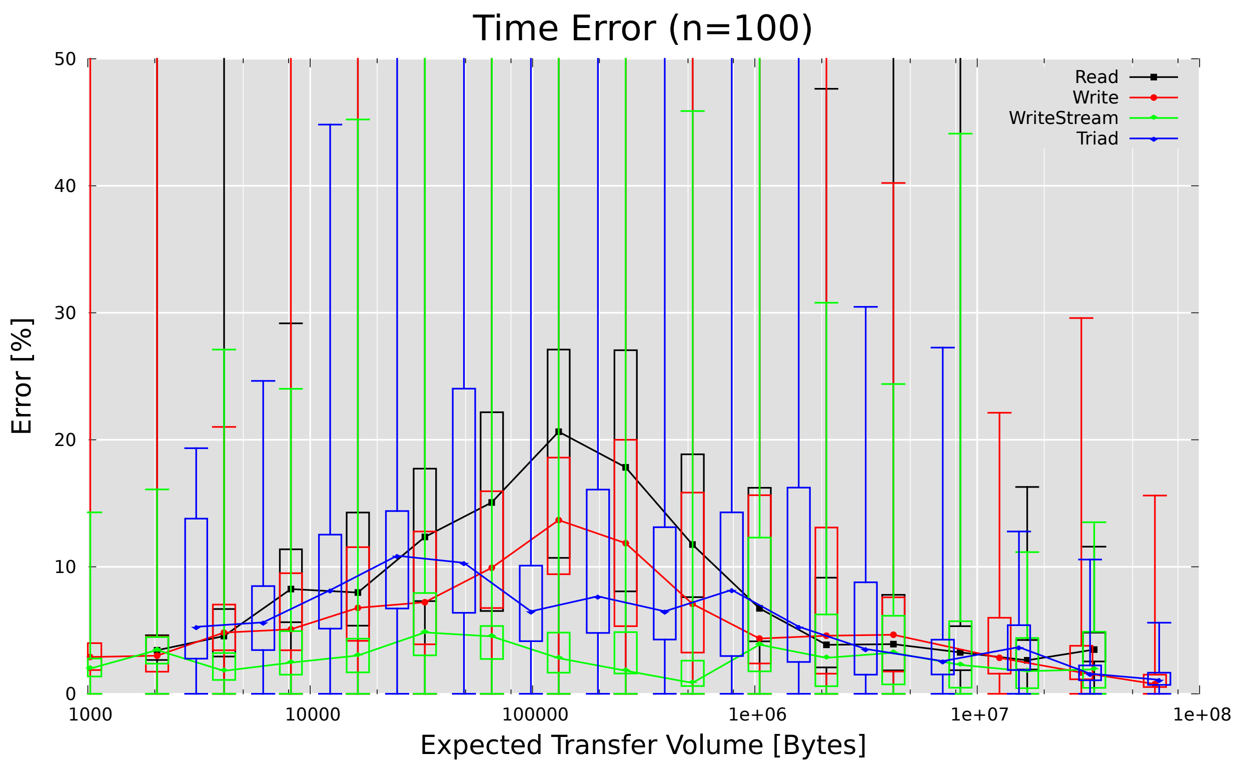


Figure C.8: Core: Error of the Execution Time of the Memory Intensive Kernels: $\text{err}\left(\frac{\text{Execution Time}}{\min(\text{Execution Time})}\right)$

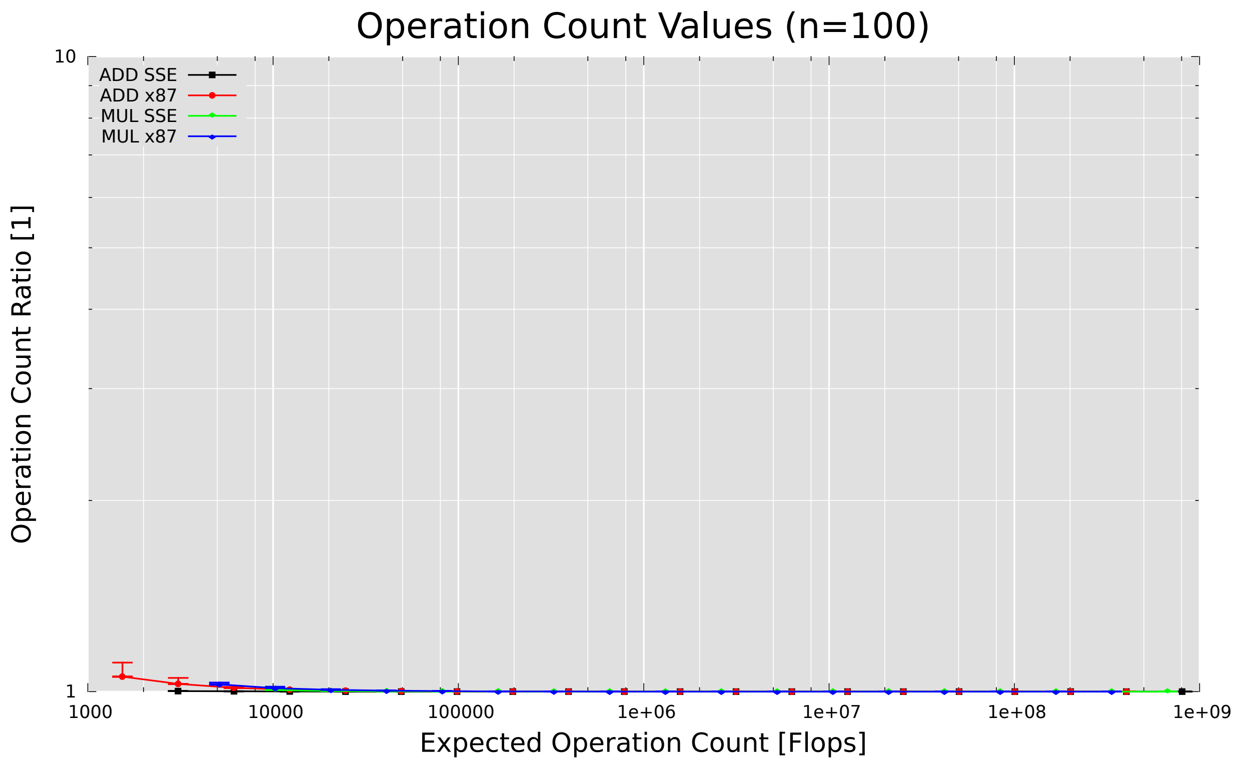
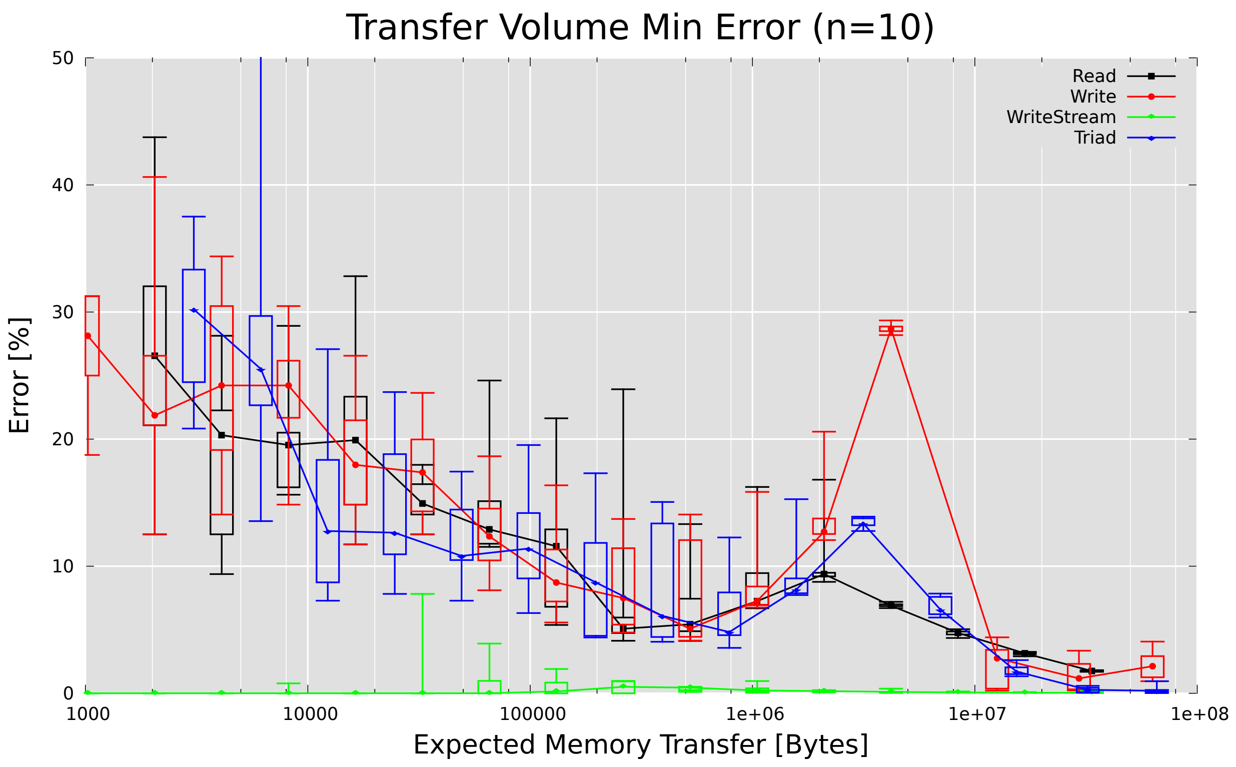
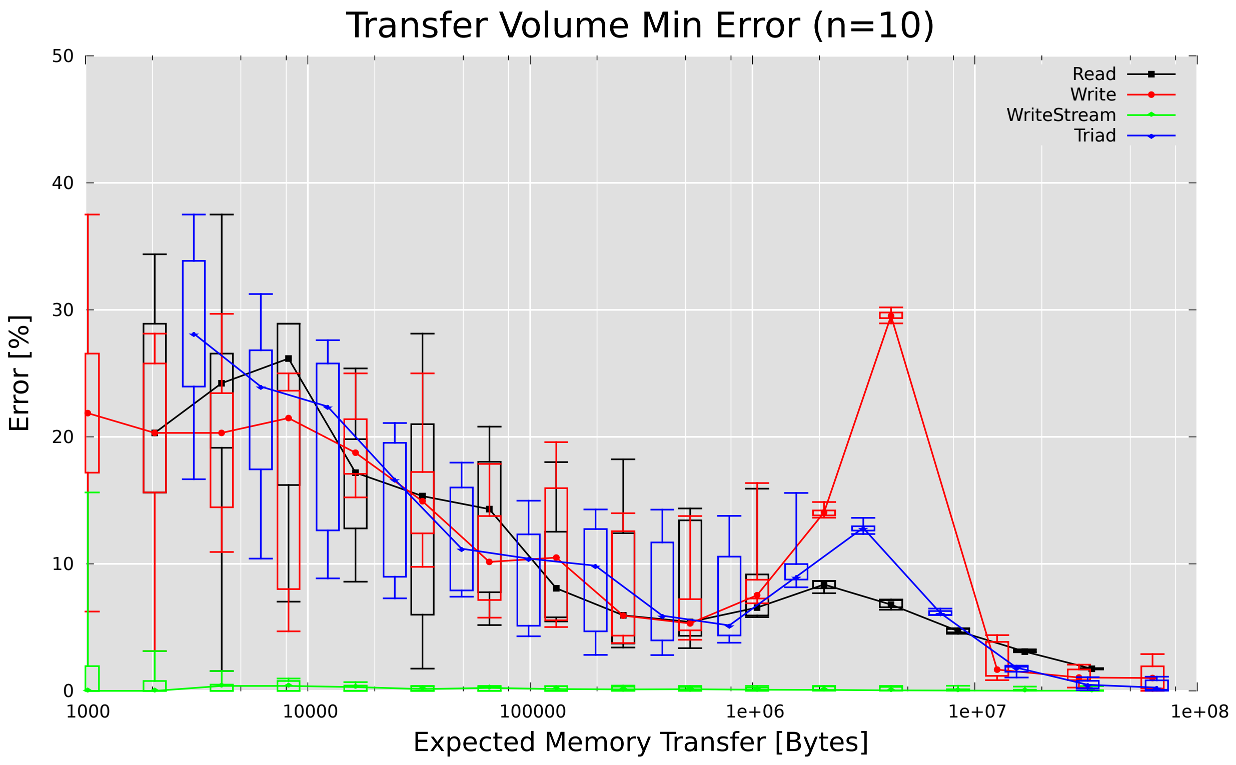


Figure C.9: Core: Operation Count: $\frac{\text{Actual Operation Count}}{\text{Expected Operation Count}}$



(a) MemBus



(b) MemL2

Figure C.10: Core: Error of the Transfer Volume with $K = 10$: $\mathrm{err}\left(\frac{\tau_{10}}{\Theta}\right)$

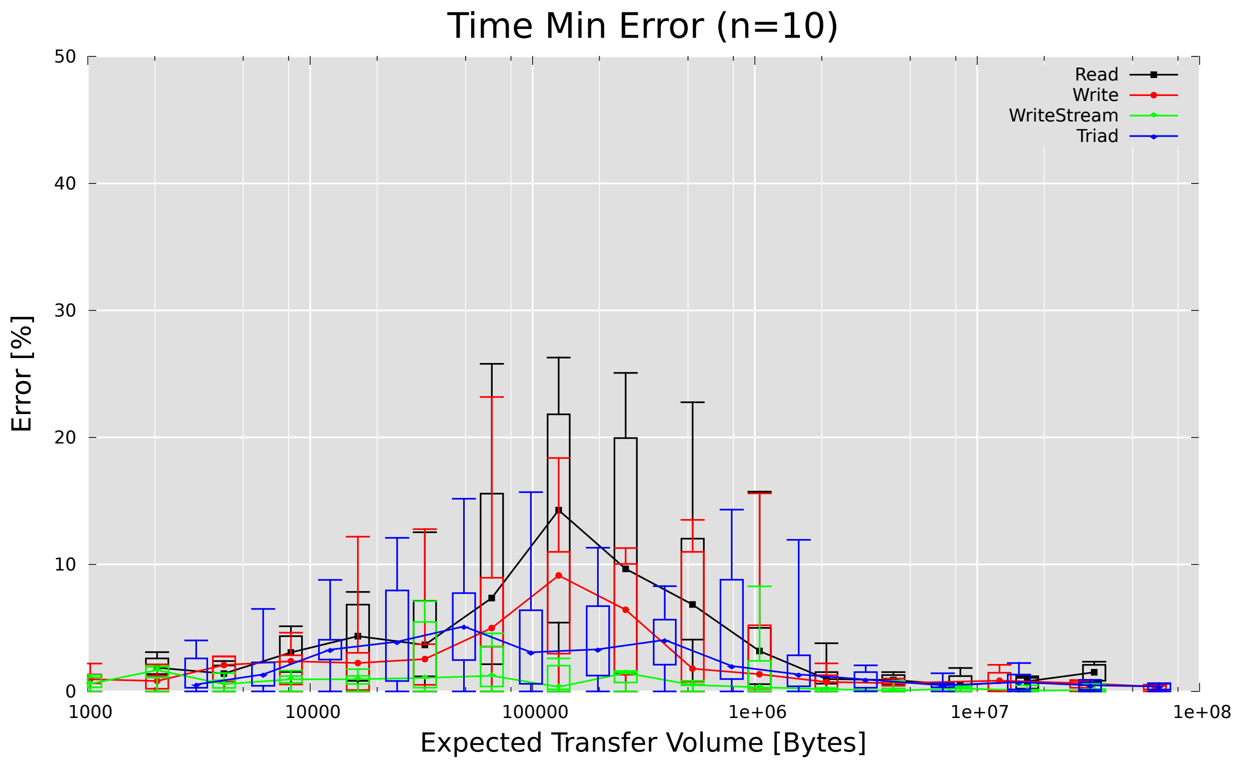


Figure C.11: Core: Error of the Execution Time with $K = 10$: err $\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

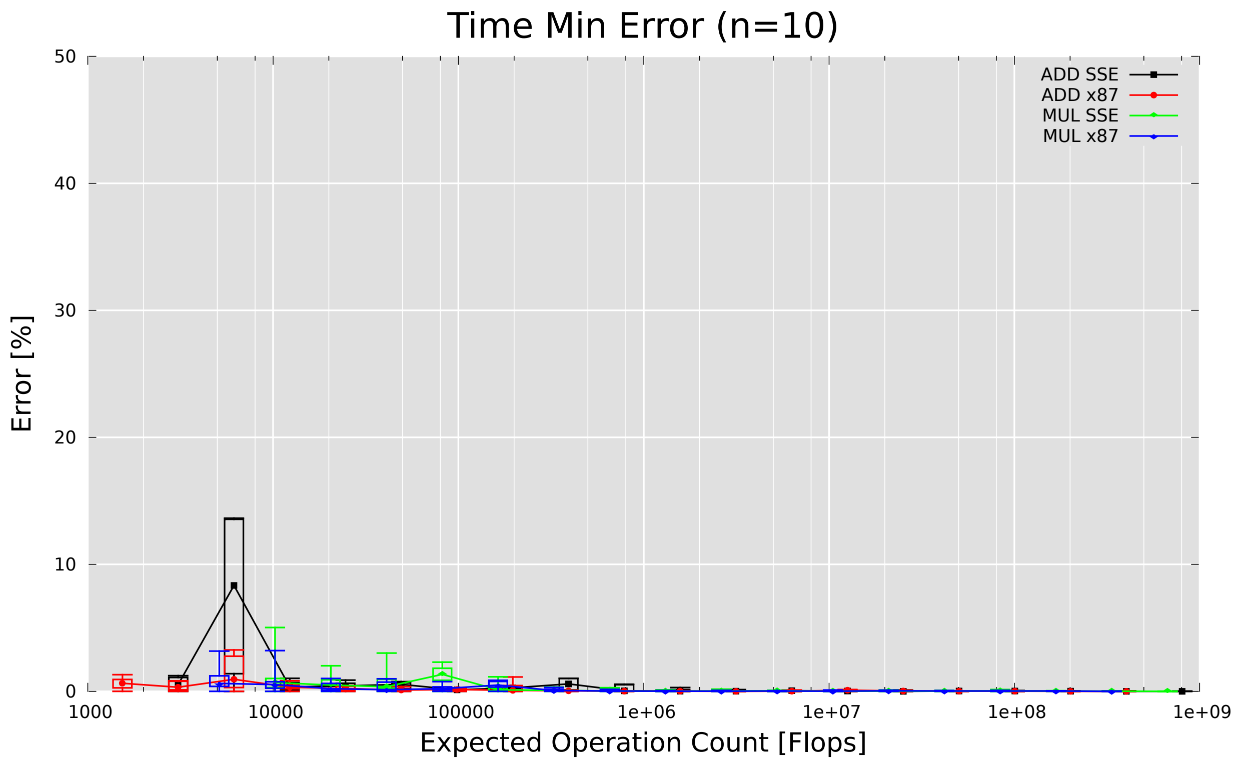


Figure C.12: Core: Error of Execution Time with $K = 10$: err $\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

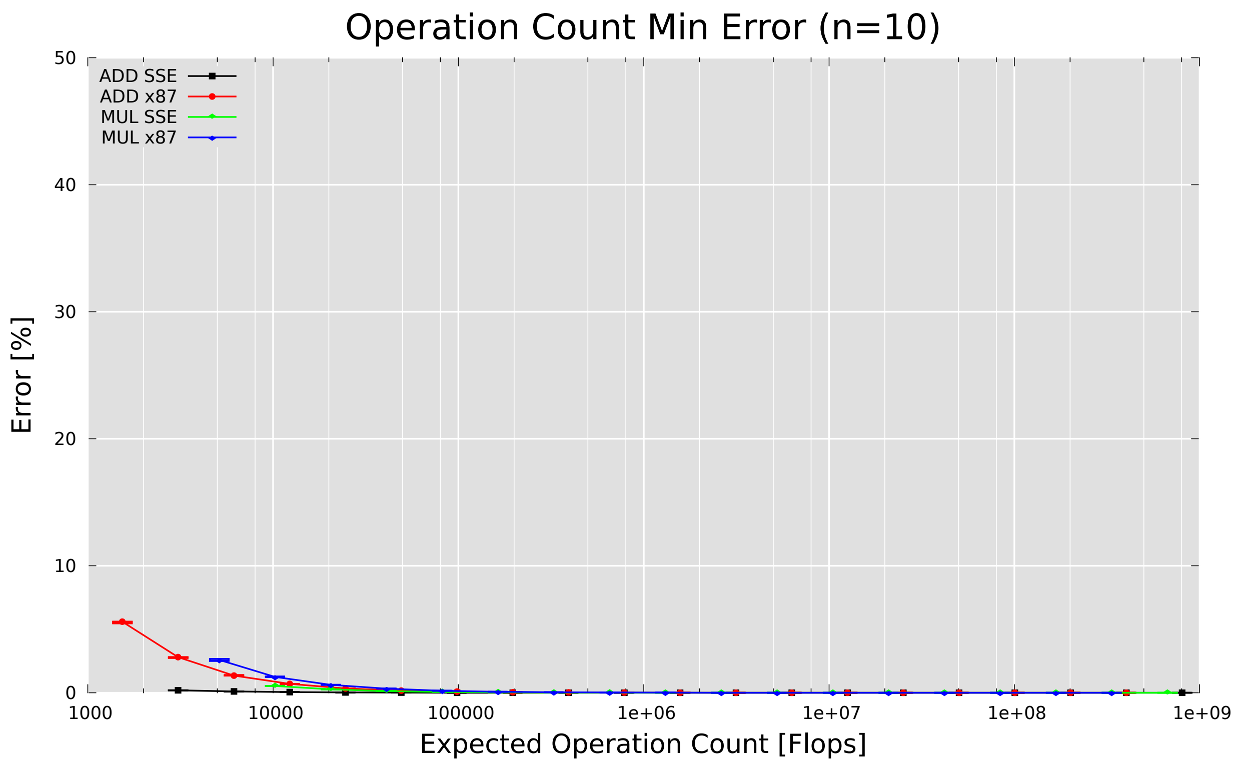
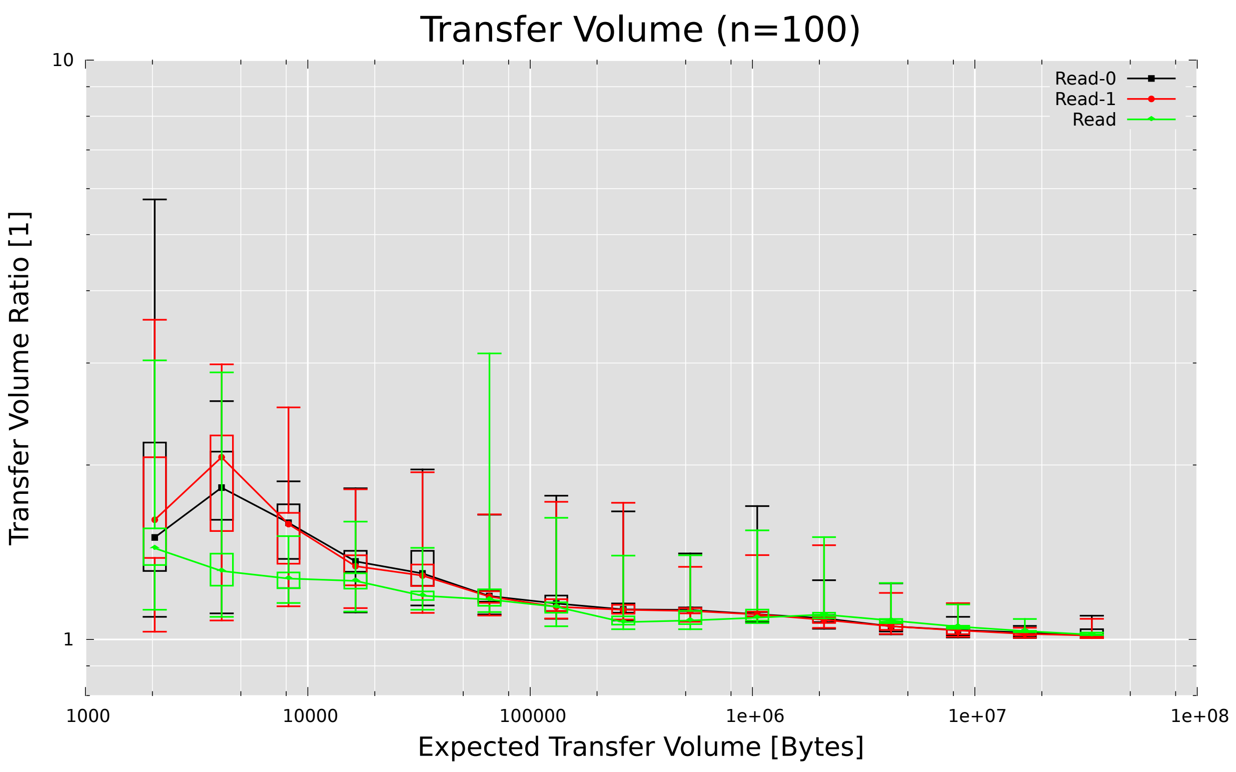
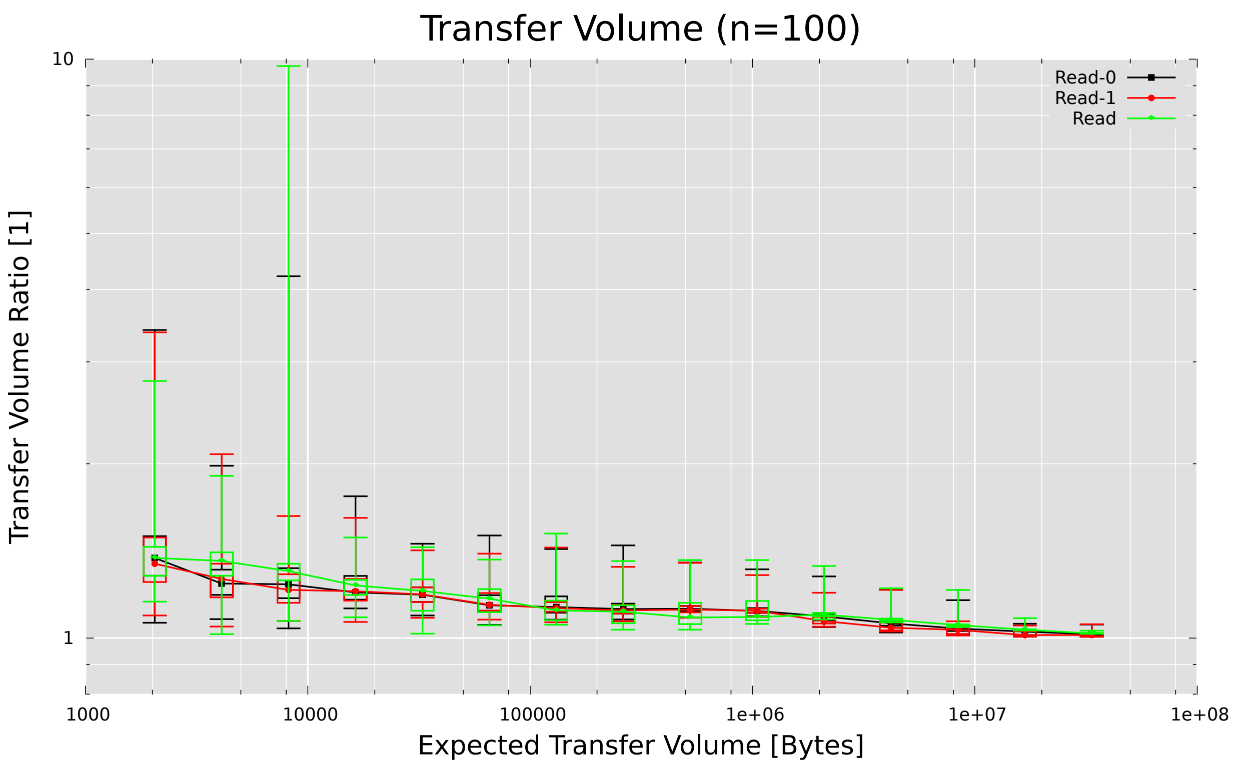


Figure C.13: Core: Error of Operation Count with $K = 10$: $\mathrm{err}(\frac{\text{Actual Operation Count}_{10}}{\text{Expected Operation Count}})$

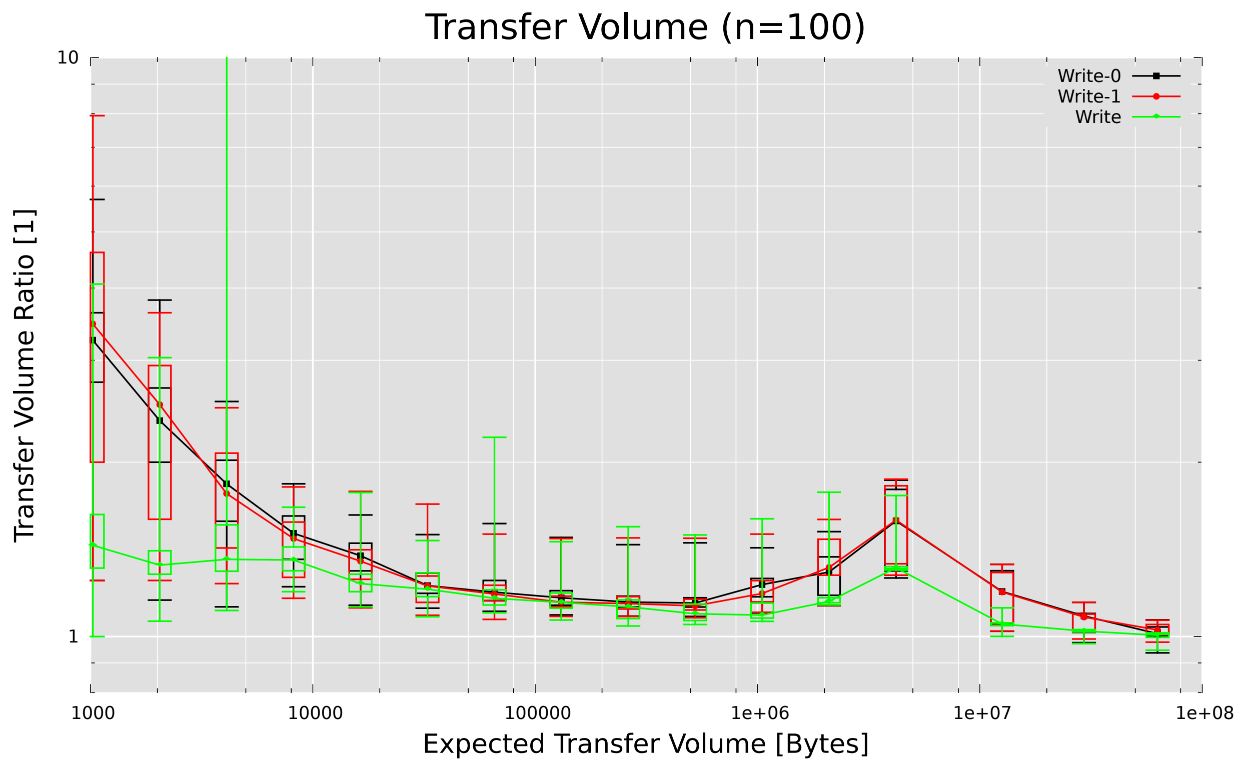


(a) MemBus

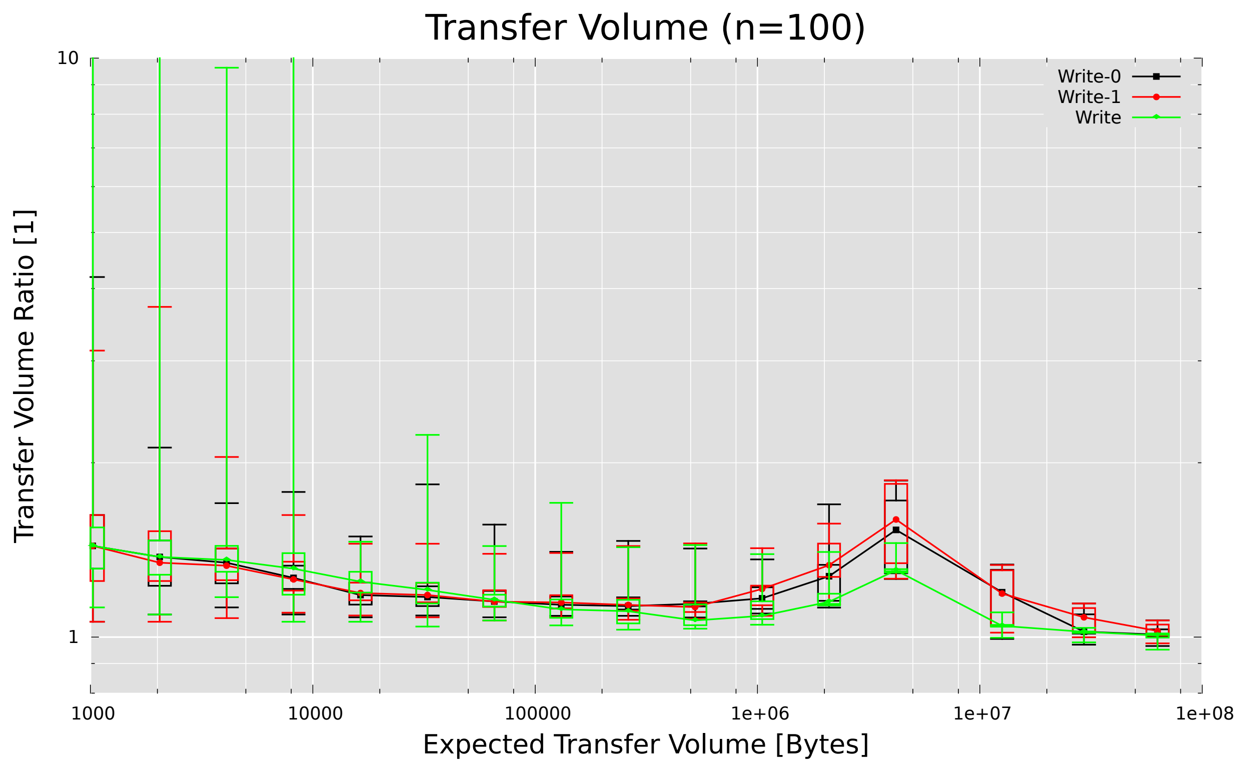


(b) MemL2

Figure C.14: Core: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Read Kernels running in Parallel (using one Buffer per Kernel)

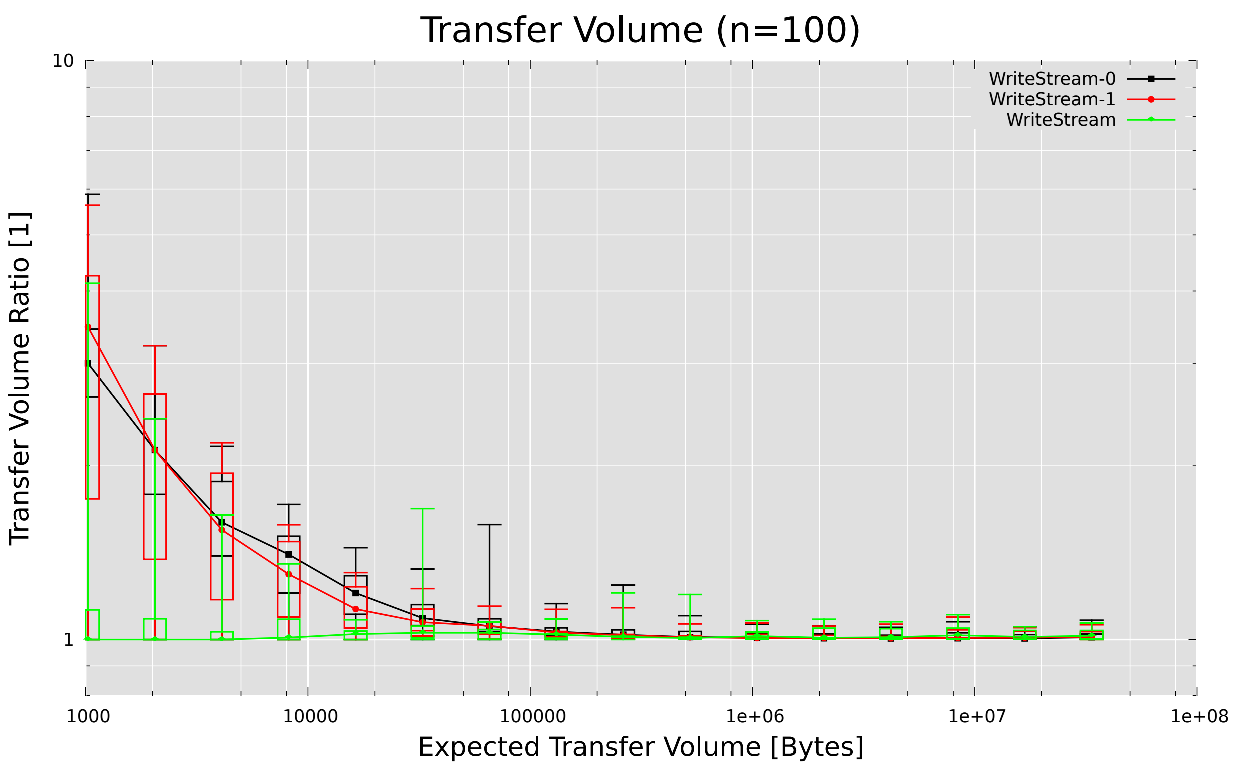


(a) MemBus

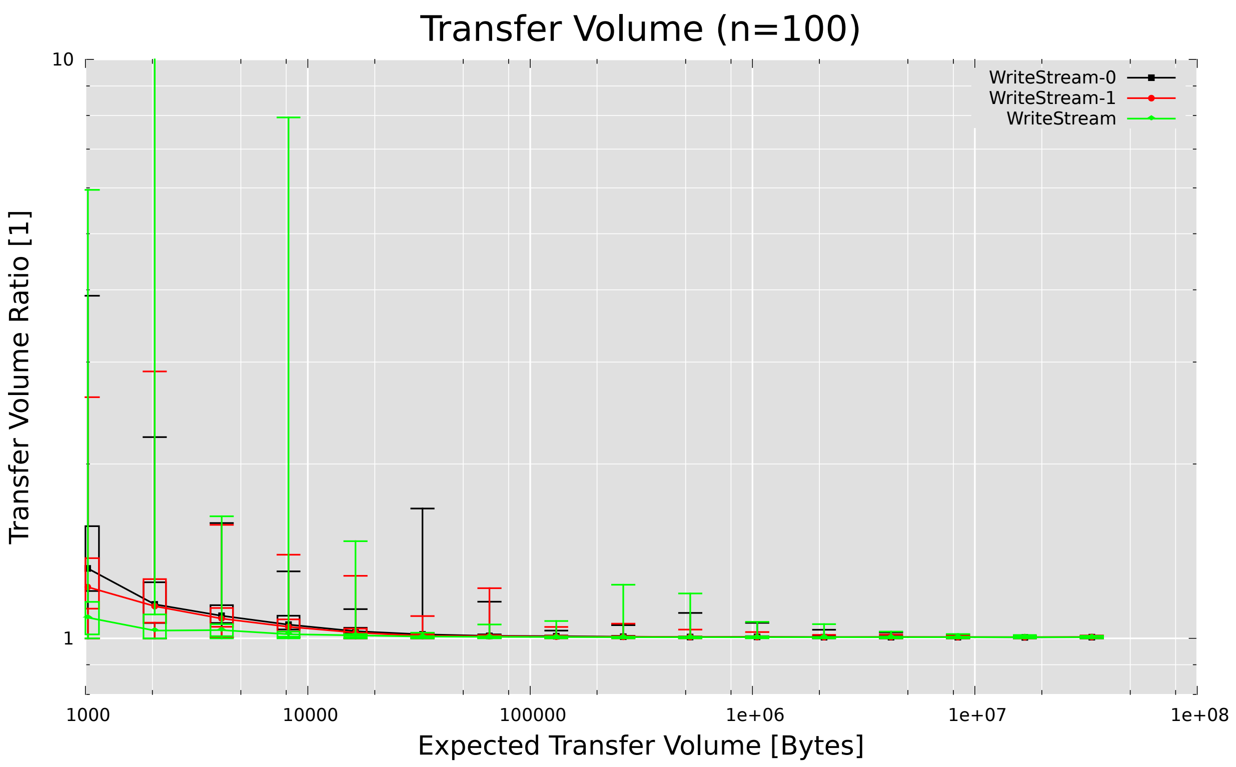


(b) MemL2

Figure C.15: Core: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Write Kernels running in Parallel (using one Buffer per Kernel)

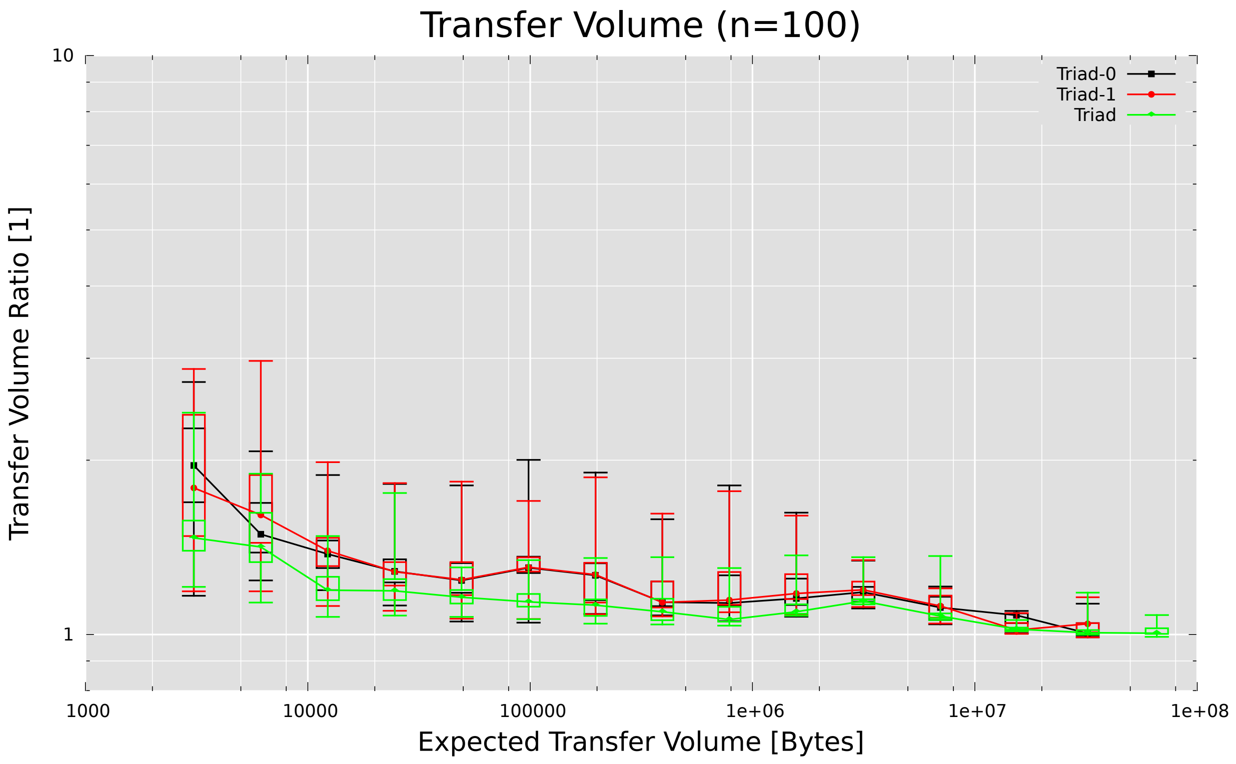


(a) MemBus

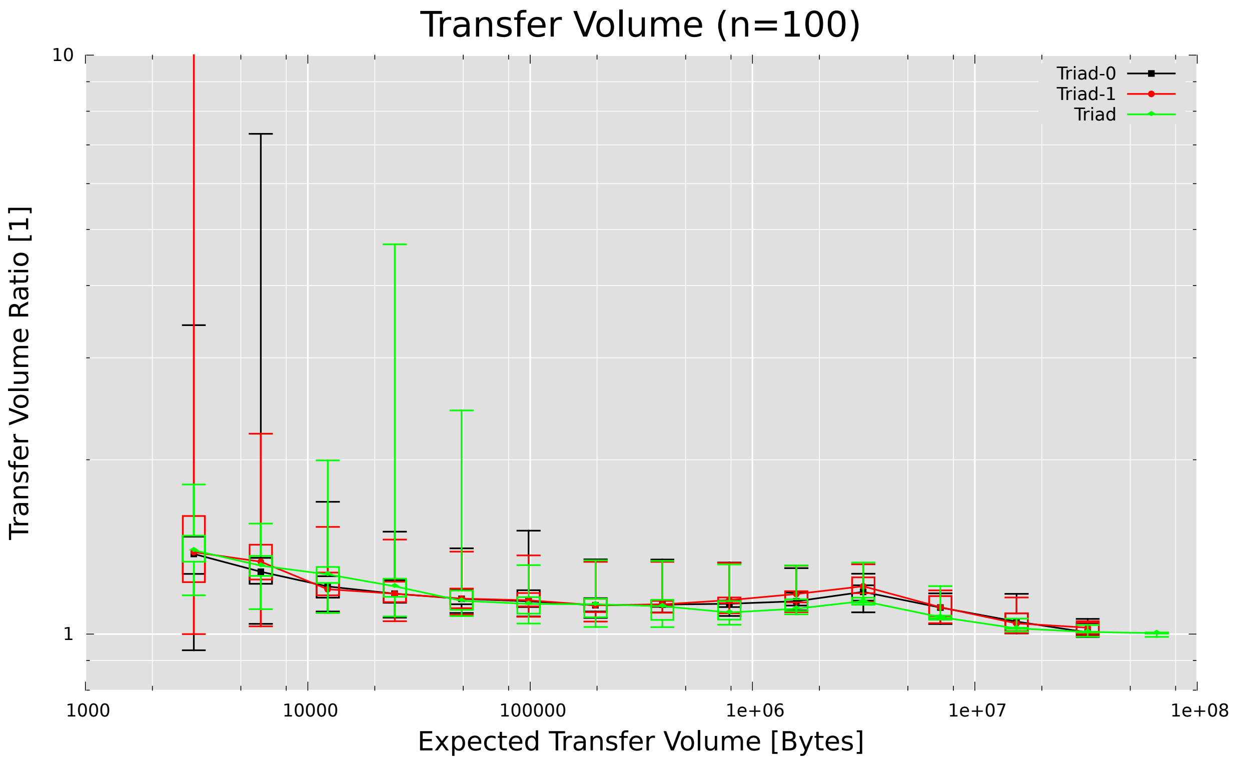


(b) MemL2

Figure C.16: Core: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Write Kernels using Streaming Stores running in Parallel (using one Buffer per Kernel)



(a) MemBus



(b) MemL2

Figure C.17: Core: Ratio of the Actual Transfer Volume to the Expected Transfer Volume for two Triad Kernels running in Parallel (using one Buffer per Kernel)

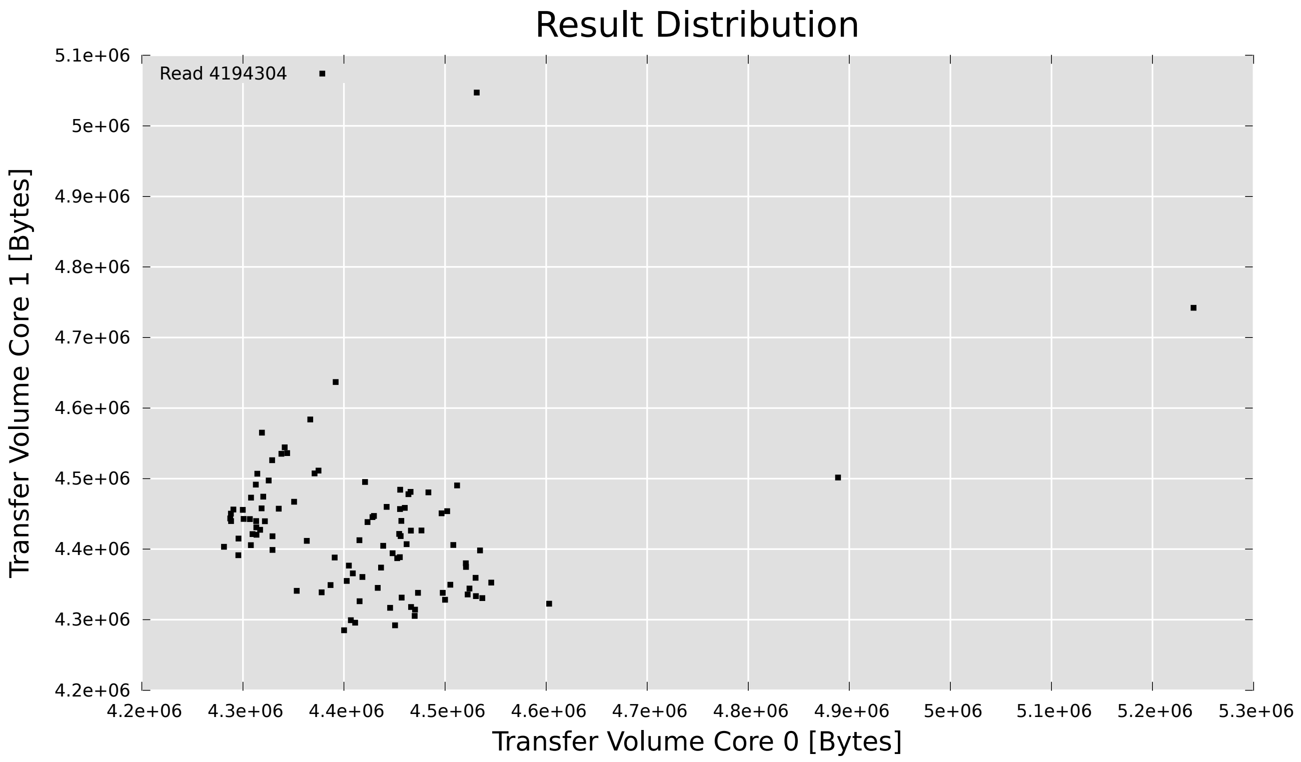


Figure C.18: Core: Distribution of the Results of the Read Kernel using MemBus

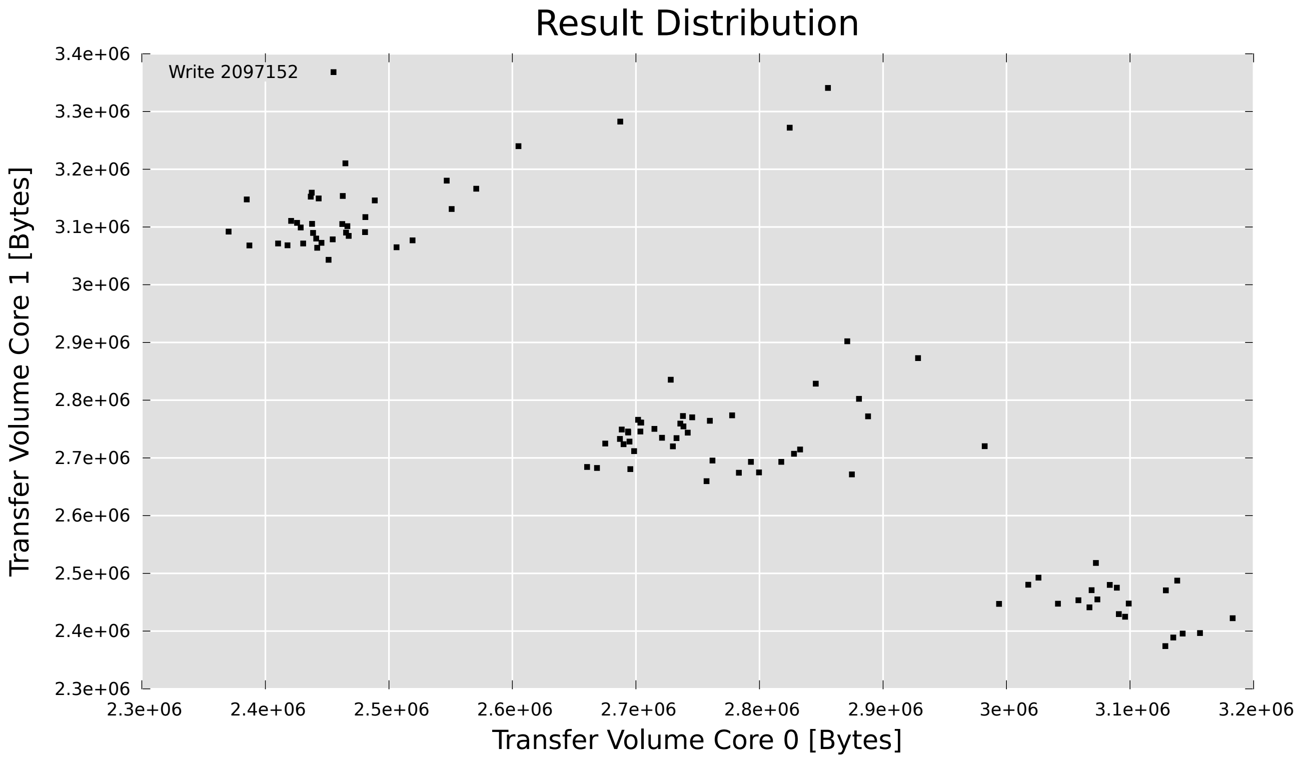


Figure C.19: Core: Distribution of the Results of the Write Kernel using MemBus

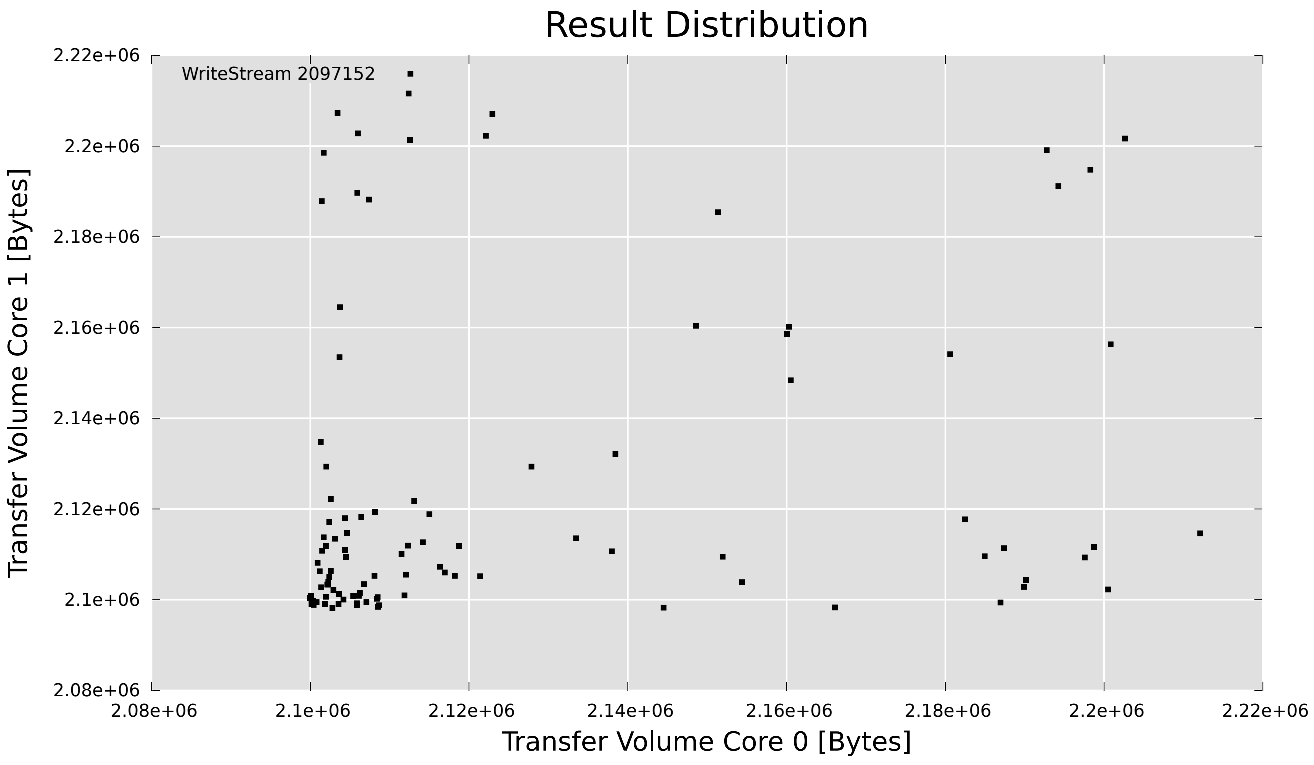


Figure C.20: Core: Distribution of the Results of the Write Kernel using Streaming Stores measured with the MemBus measurement variant

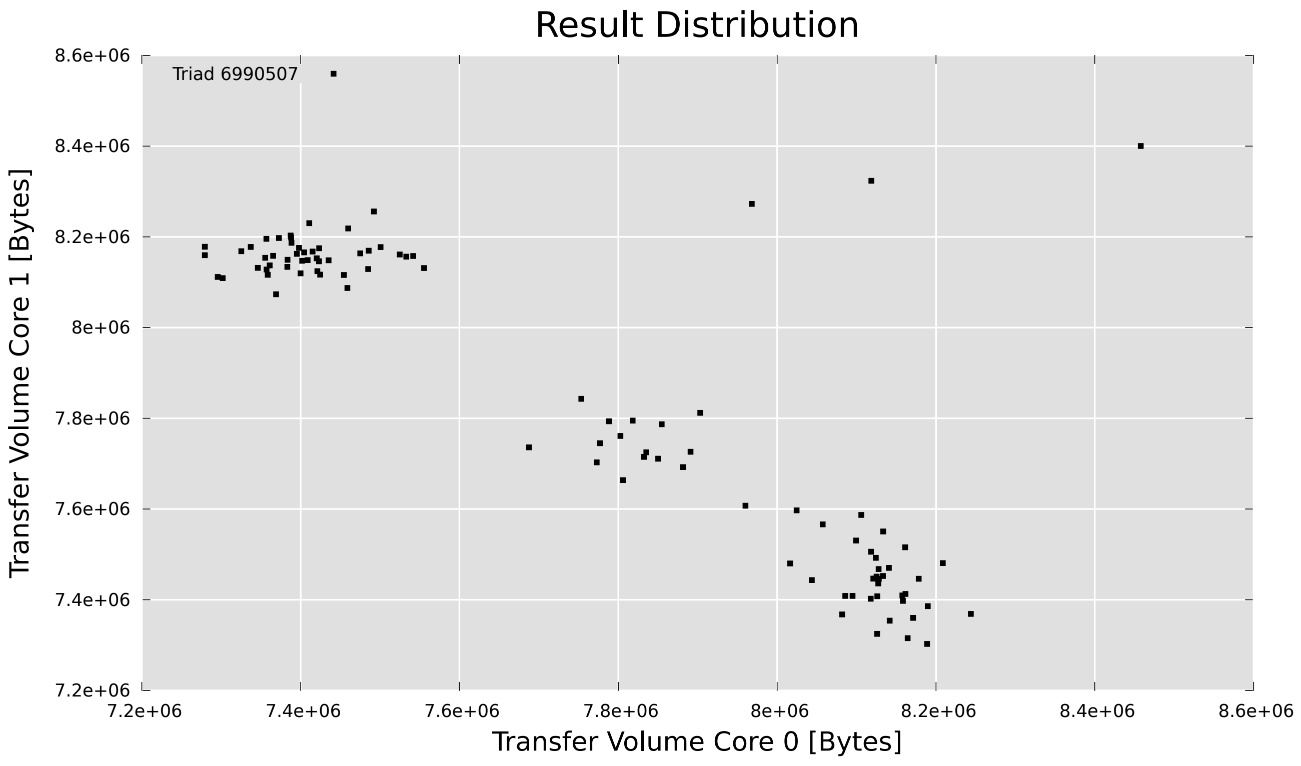


Figure C.21: Core: Distribution of the Results of the Triad Kernel using MemBus

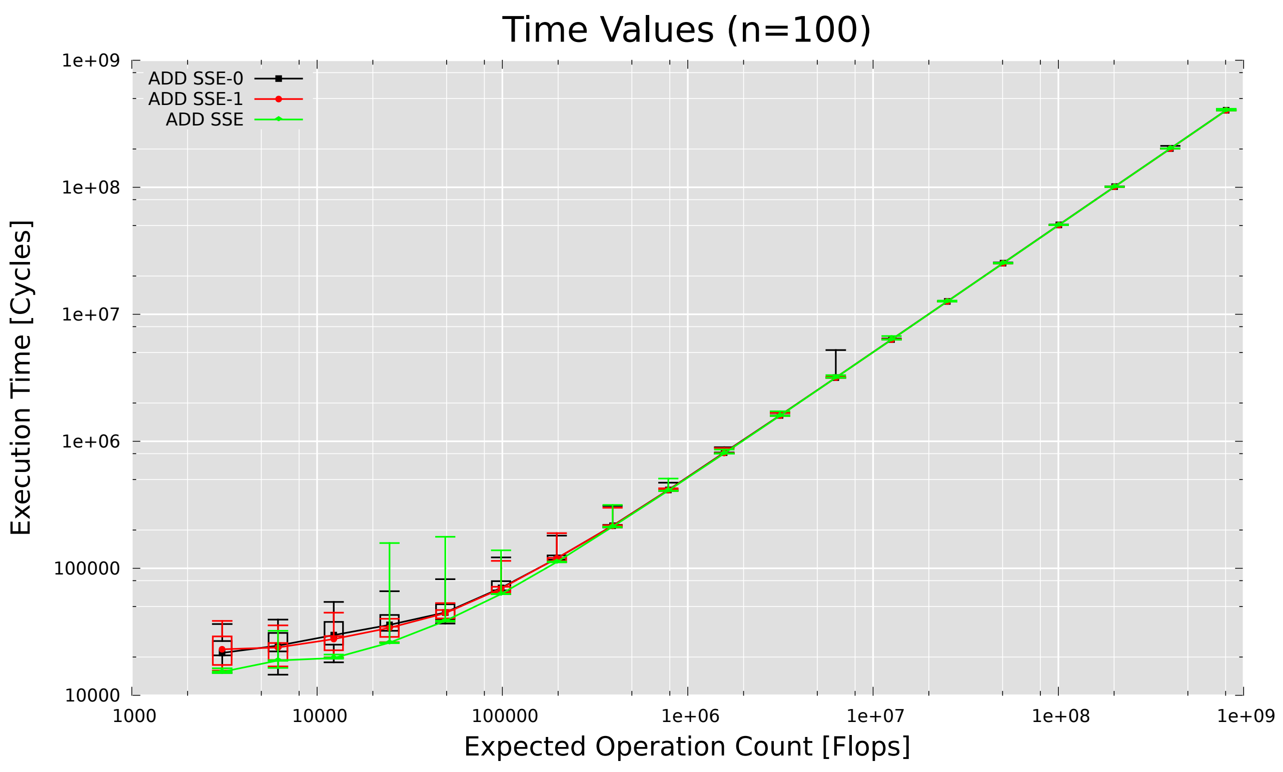


Figure C.22: Core: Execution Time of two ADD Kernels running at the same time

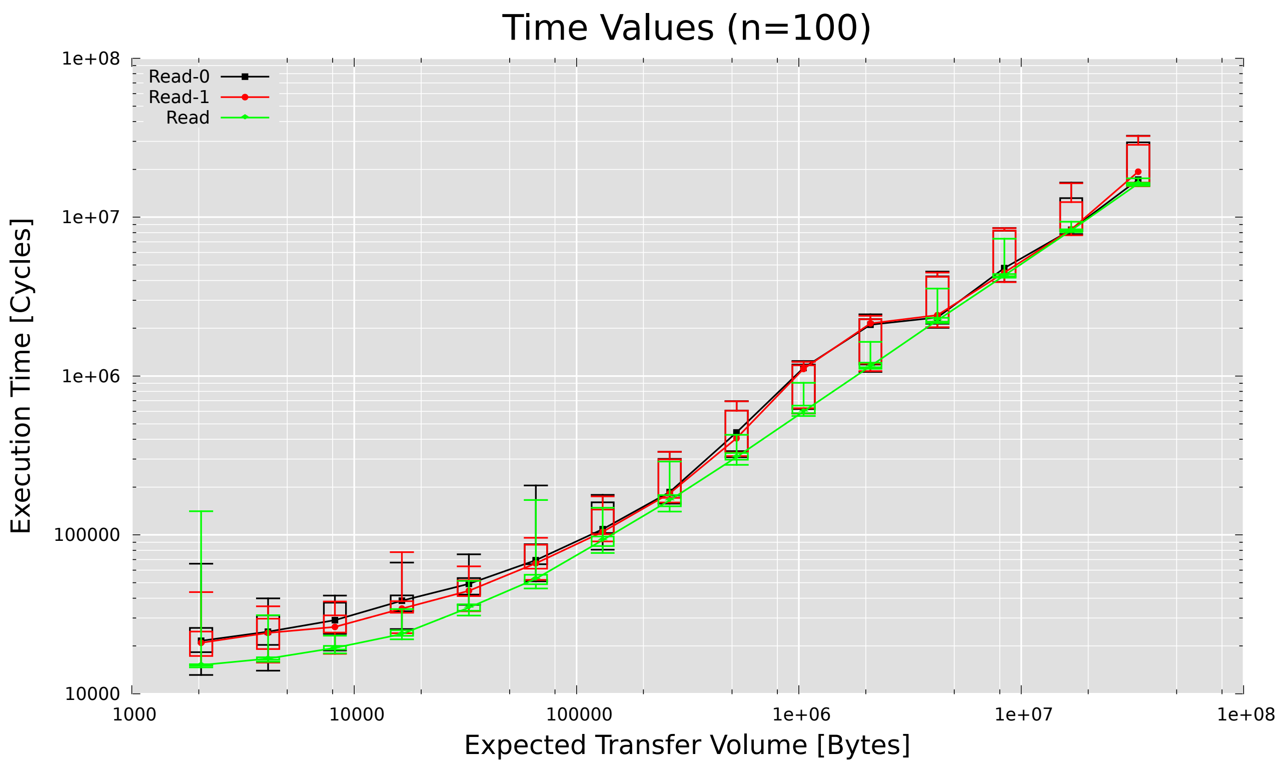


Figure C.23: Core: Execution Time of two Read Kernels running at the same time

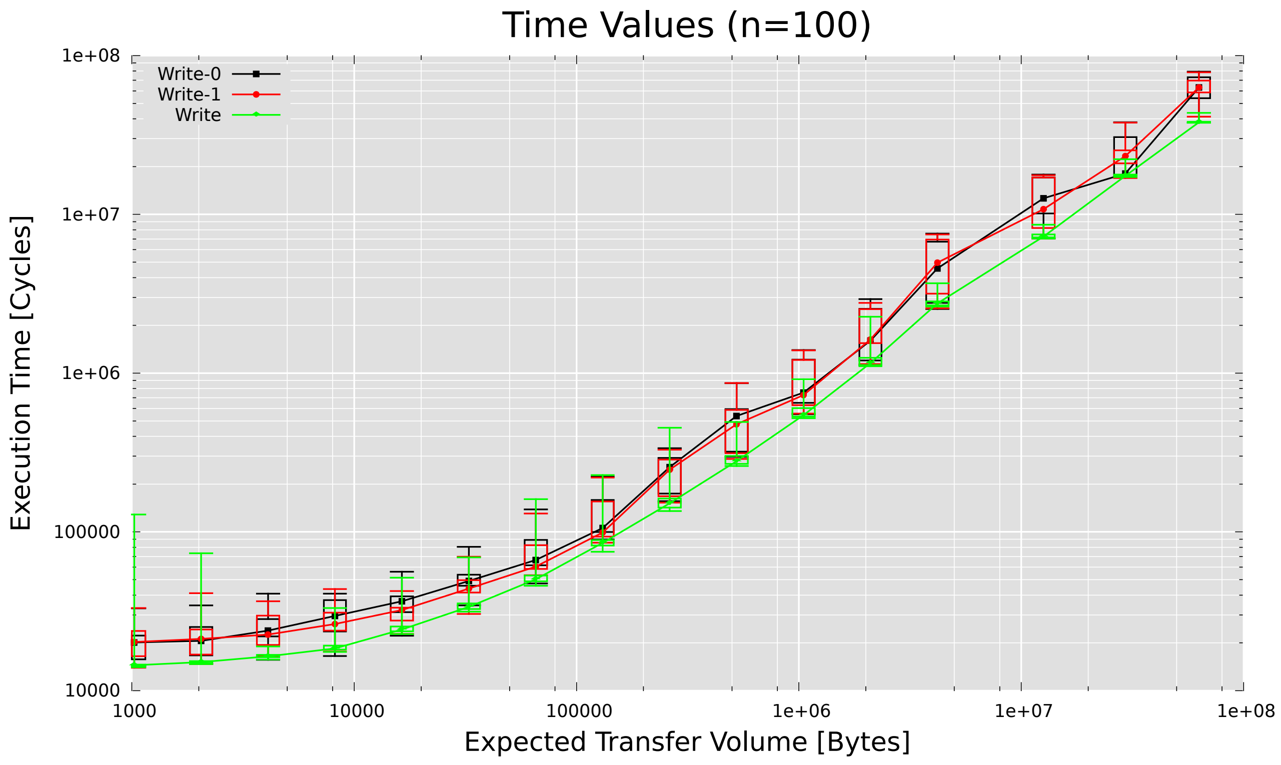


Figure C.24: Core: Execution Time of two Write Kernels running at the same time

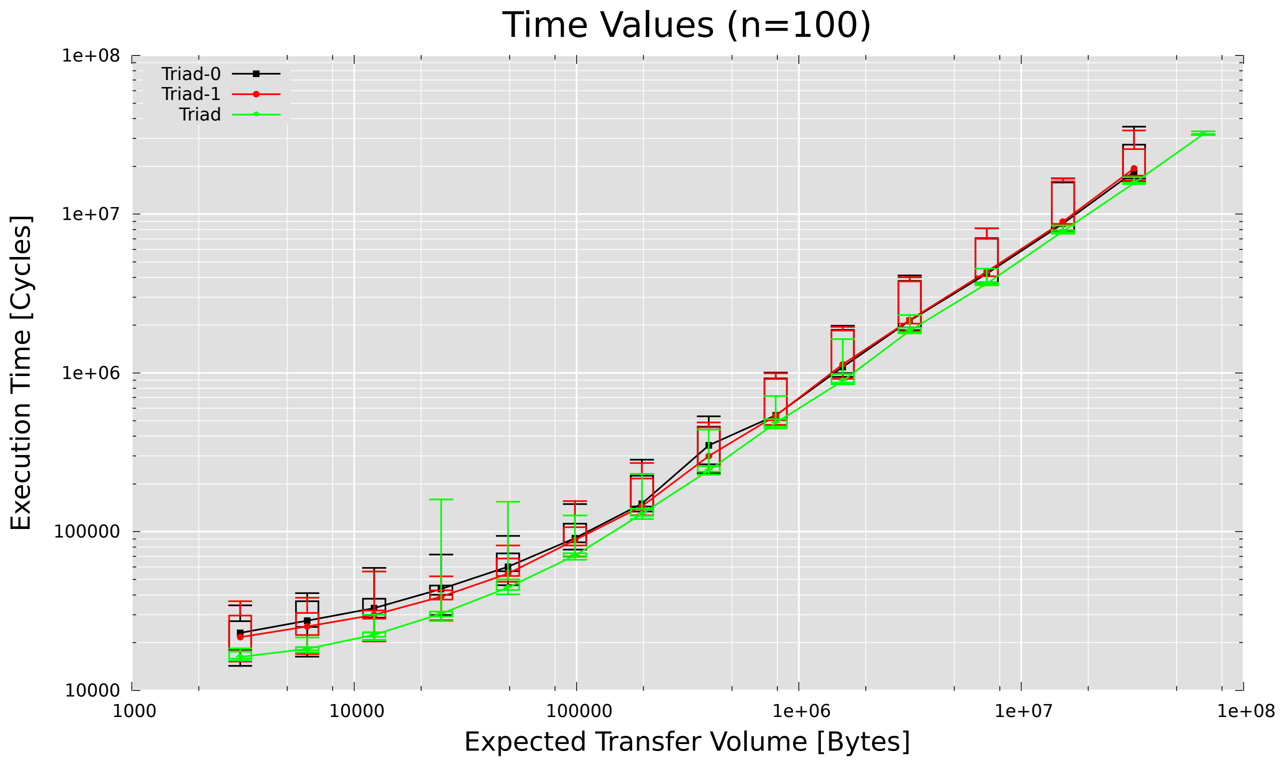


Figure C.25: Core: Execution Time of two Triad Kernels running at the same time

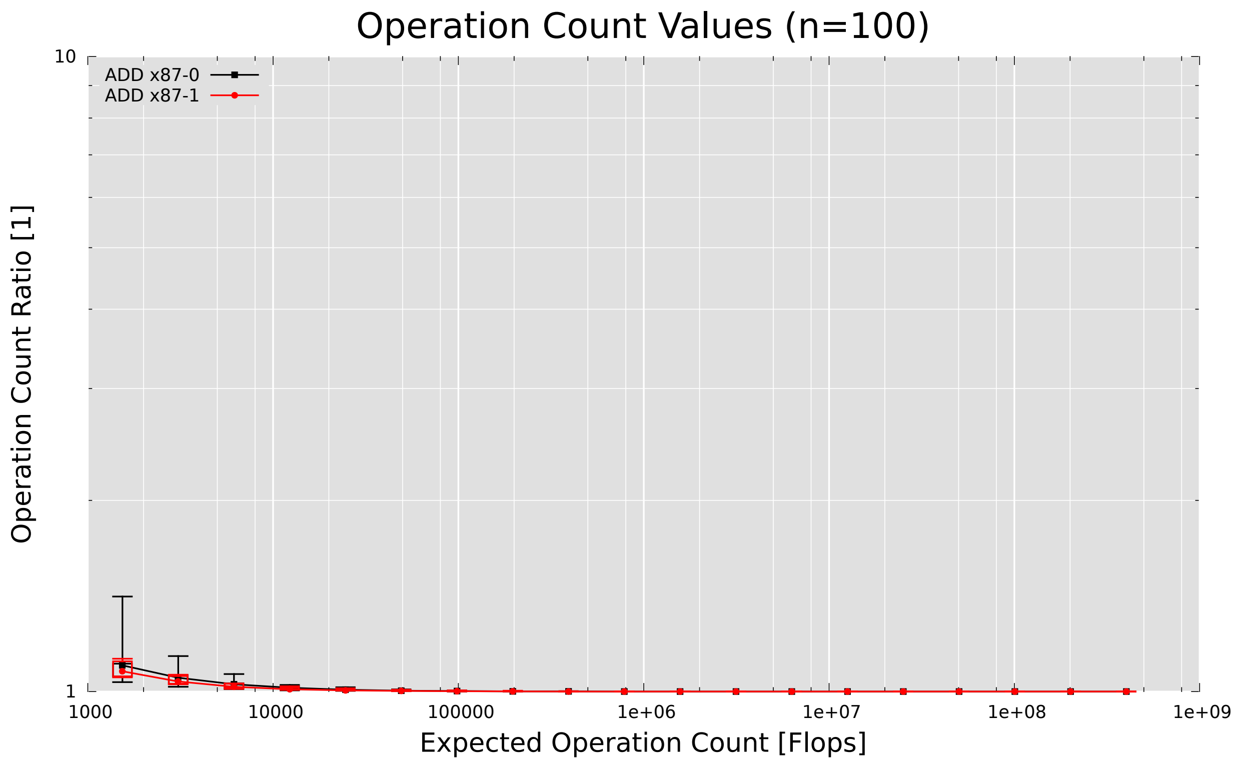
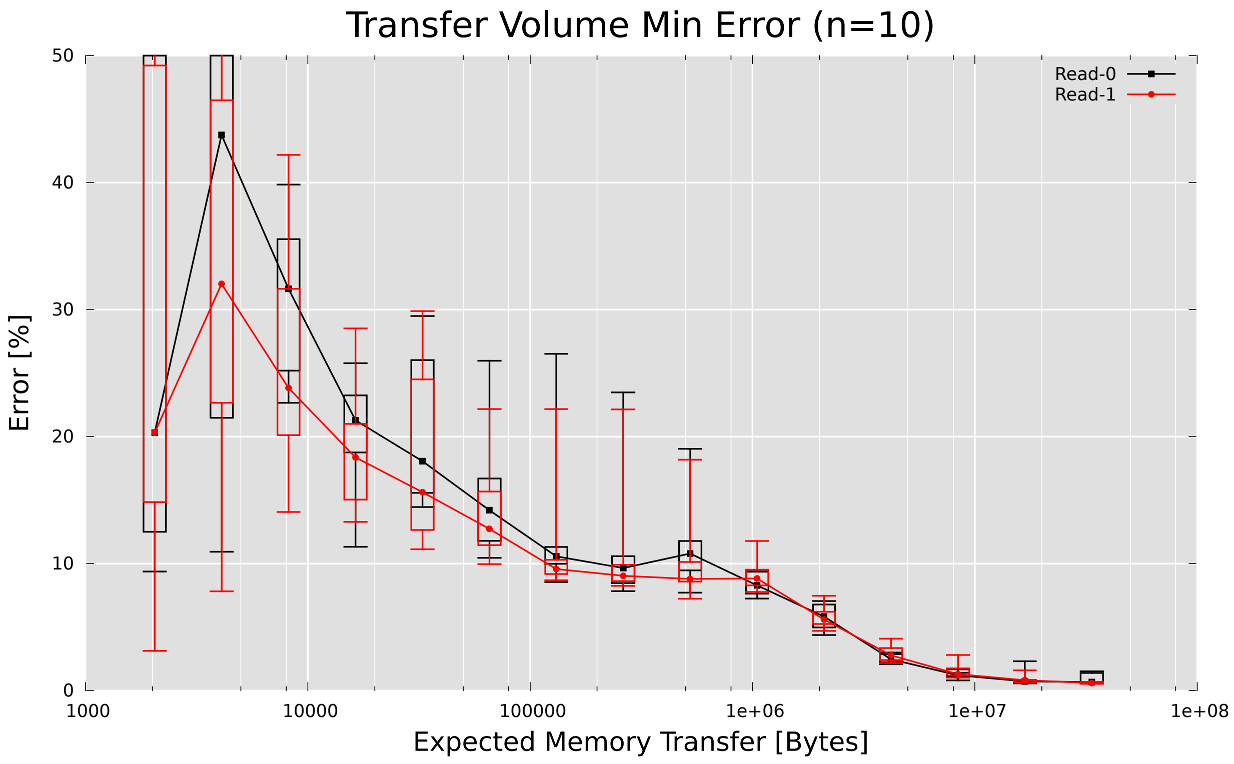
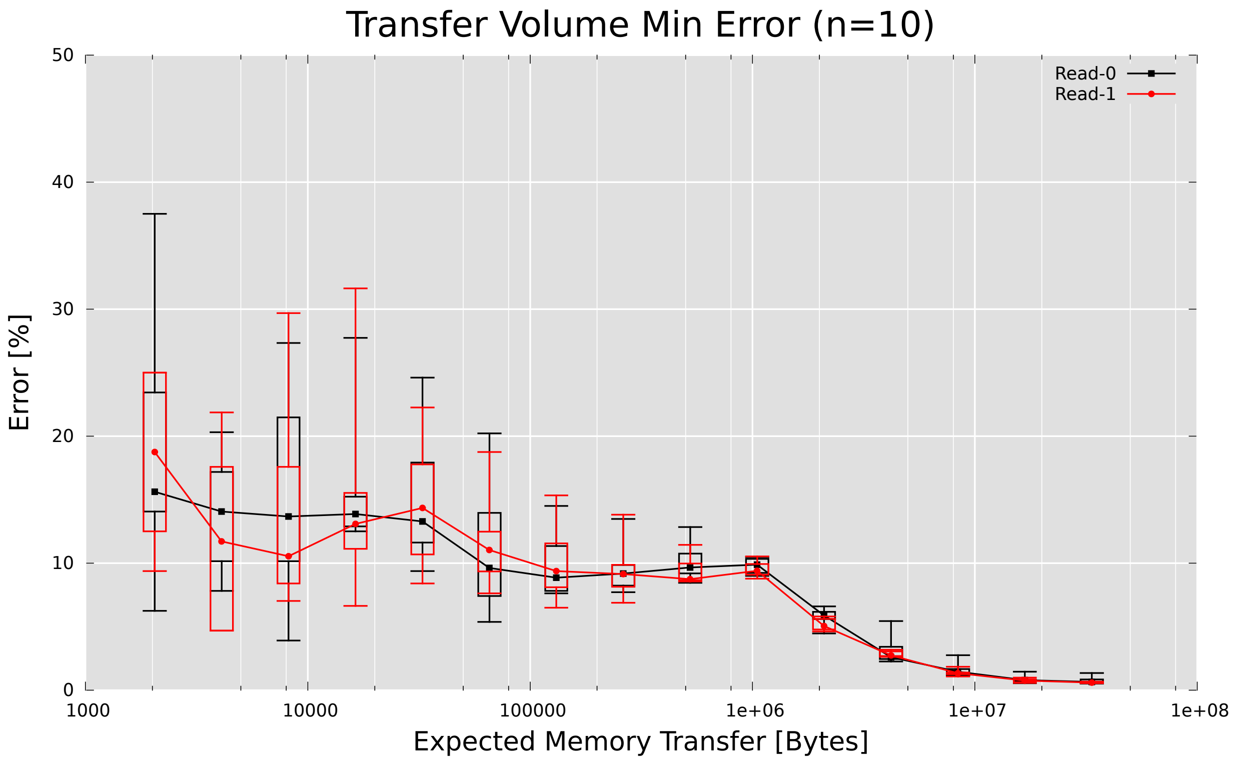


Figure C.26: Core: Ratio of the Actual Operation Count to the Expected Operation Count for two ADD Kernels running at the same time

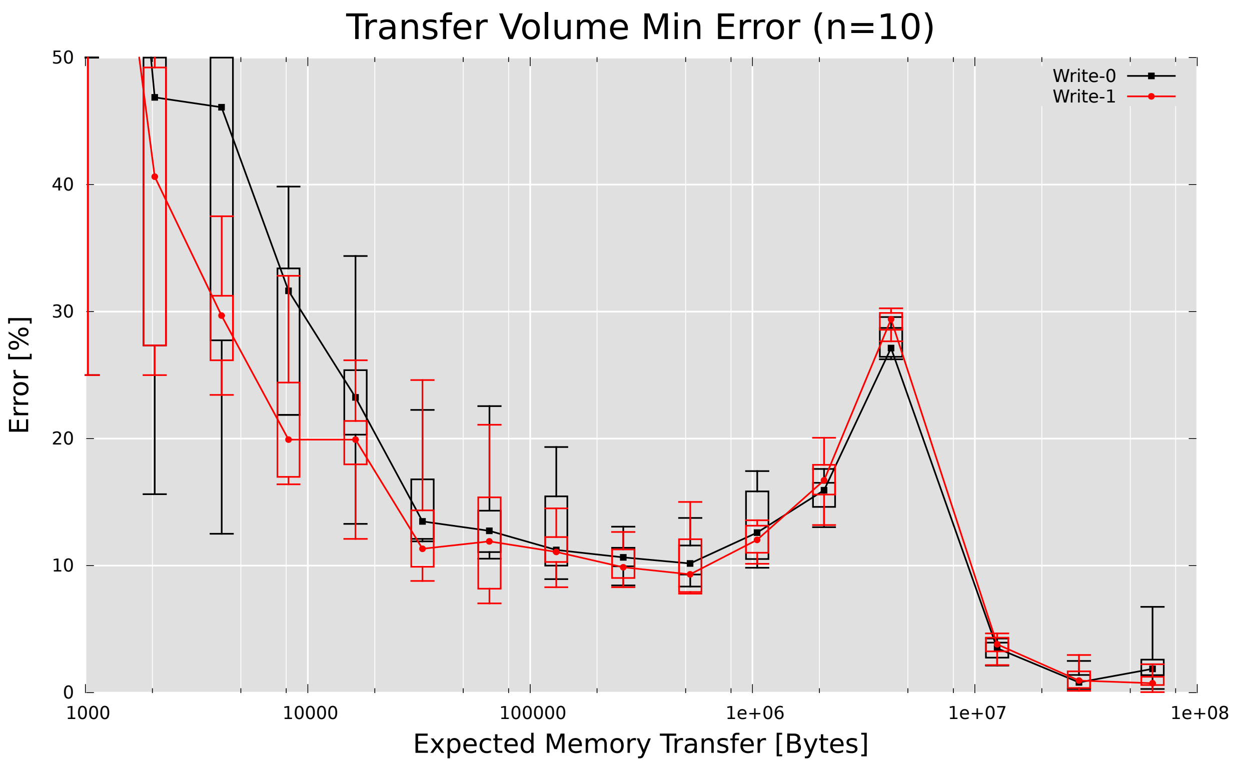


(a) MemBus

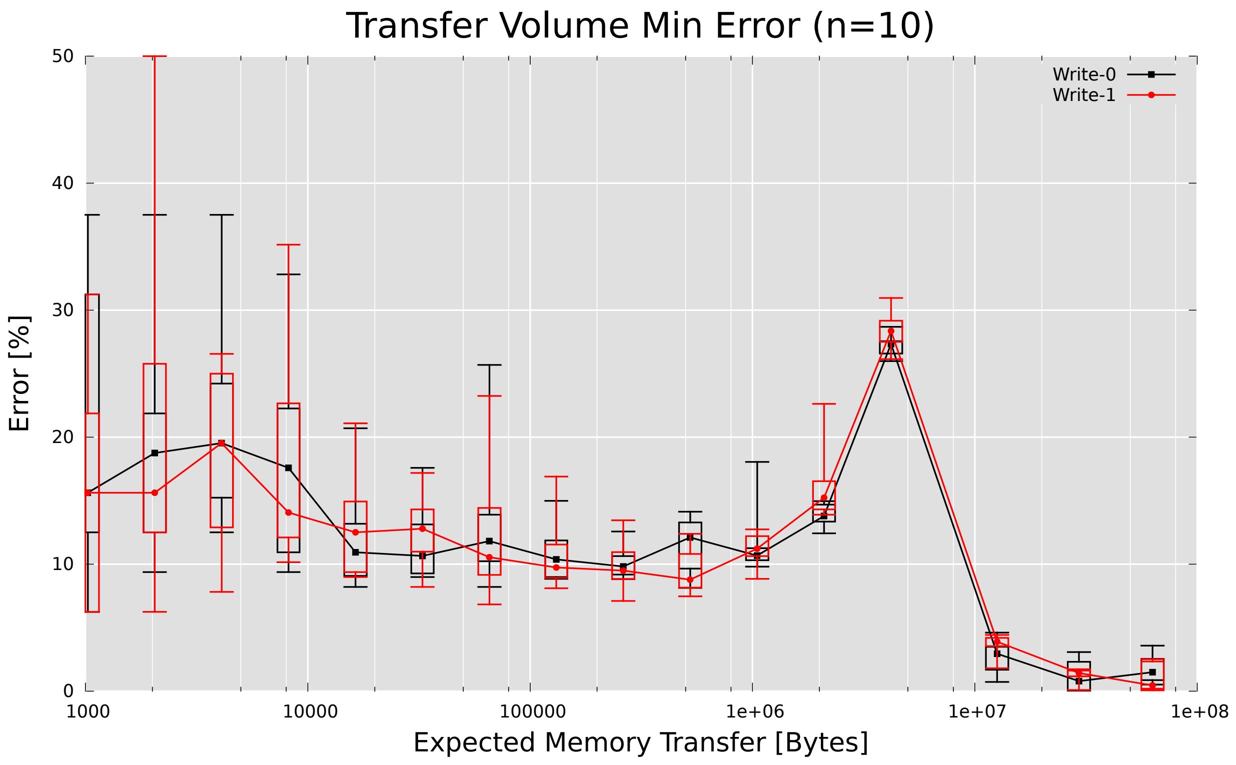


(b) MemL2

Figure C.27: Core: Error of the Transfer Volume of two Read Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\tau_{10}}{\Theta}\right)$

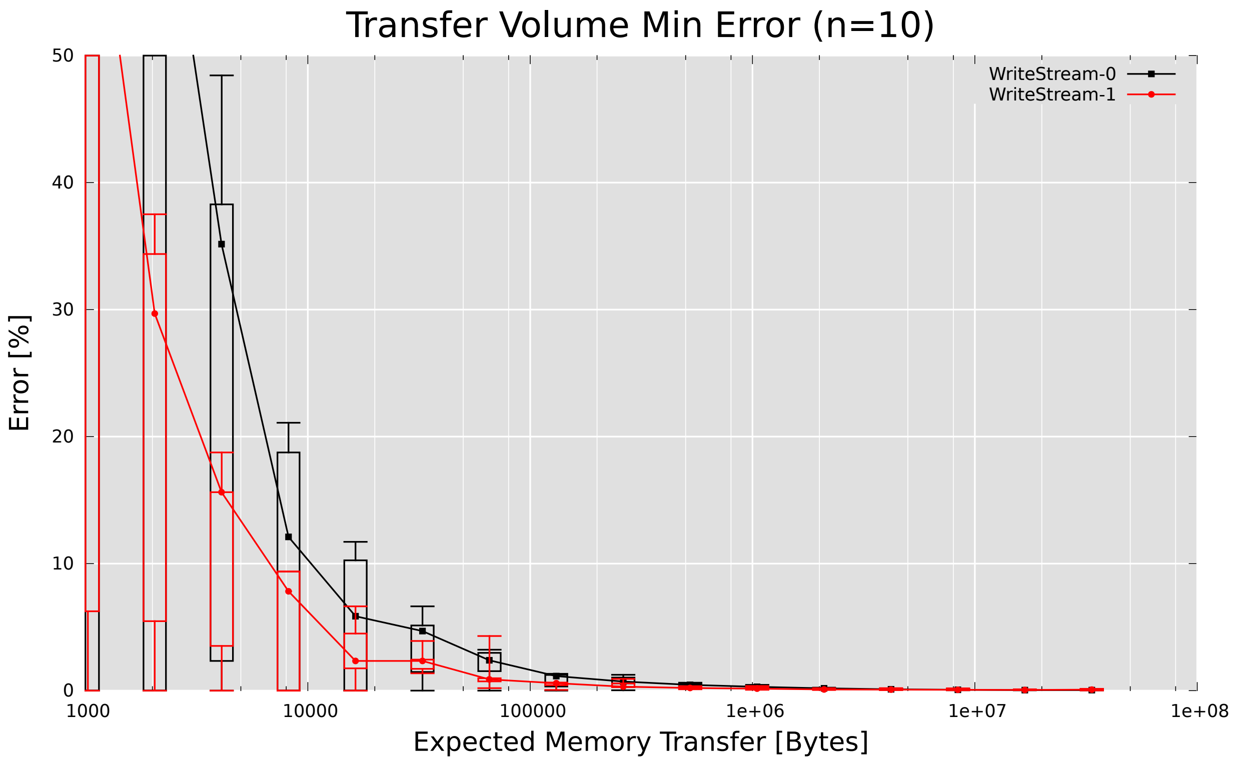


(a) MemBus

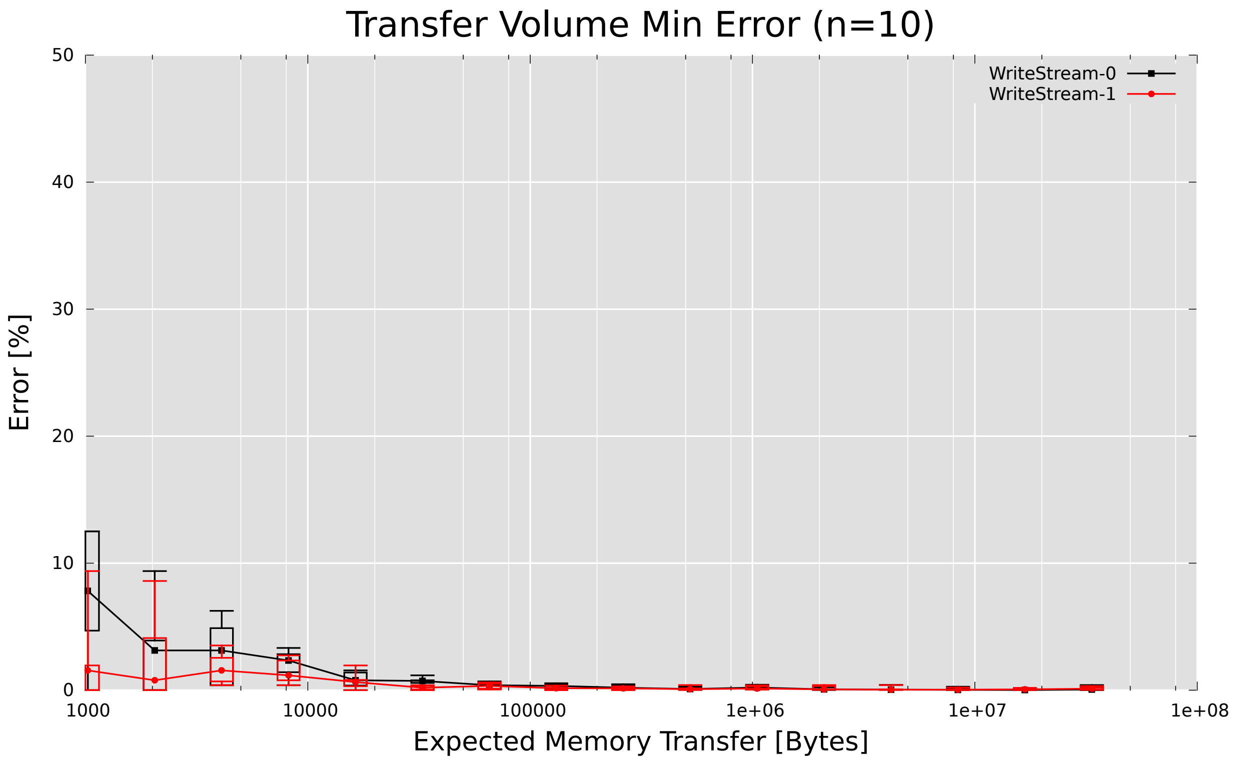


(b) MemL2

Figure C.28: Core: Error of the Transfer Volume of two Write Kernels running at the same time with $K = 10$: $\mathrm{err}\left(\frac{\tau_{10}}{\Theta}\right)$

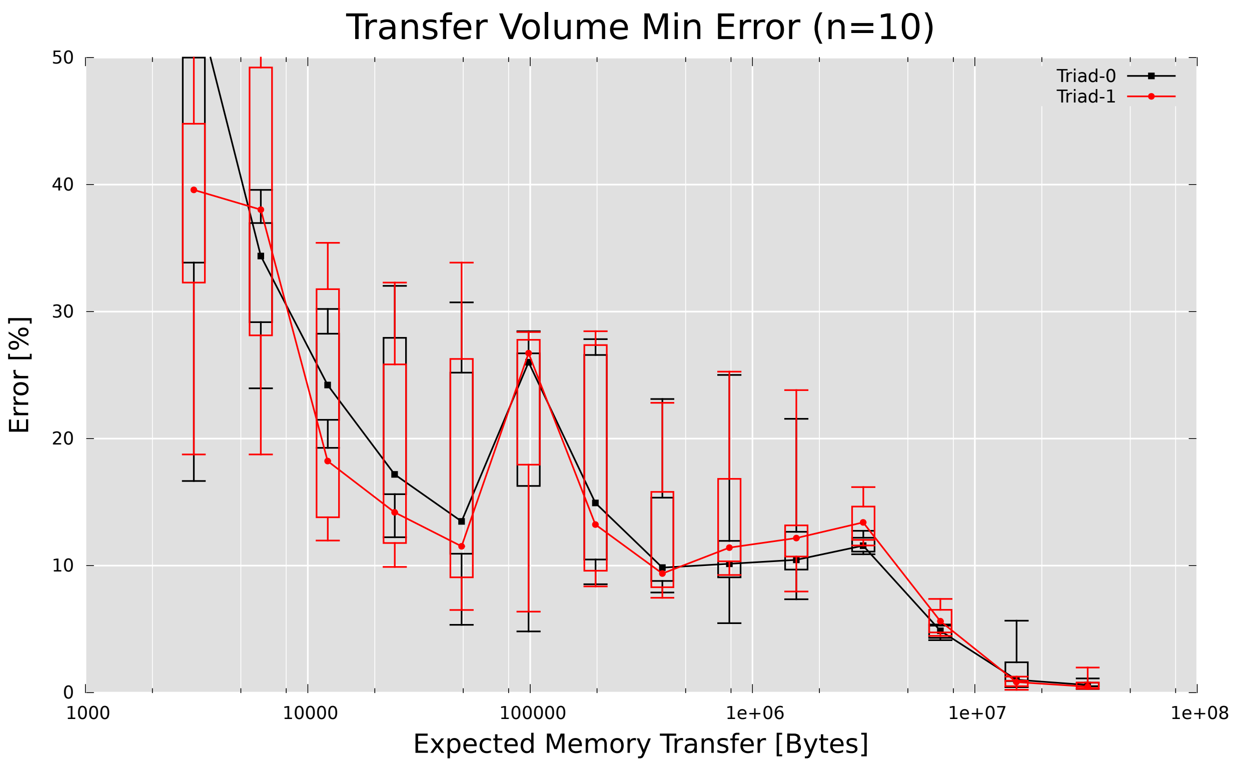


(a) MemBus

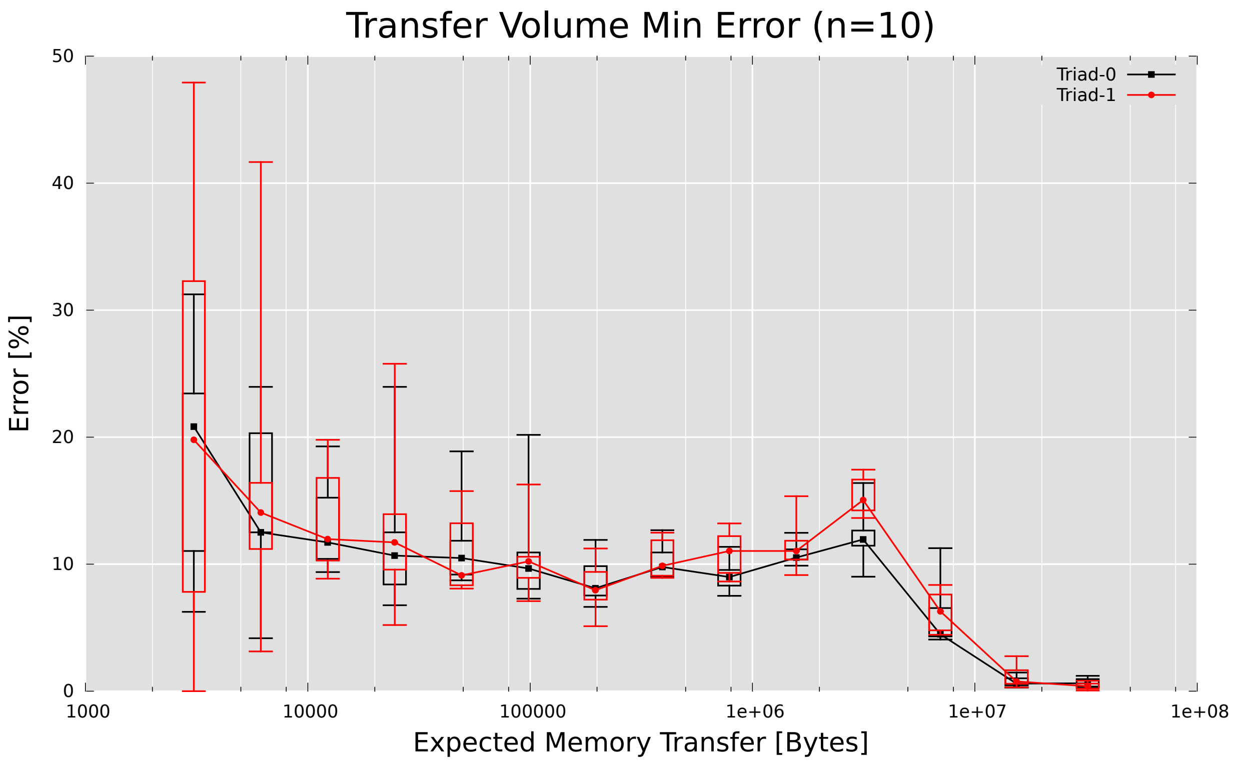


(b) MemL2

Figure C.29: Core: Error of the Transfer Volume of two Write Kernels using Streaming Stores running at the same time with $K = 10$: $\mathrm{err}\left(\frac{\tau_{10}}{\Theta}\right)$



(a) MemBus



(b) MemL2

Figure C.30: Core: Error of the Transfer Volume of two Triad Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\tau_{10}}{\Theta}\right)$

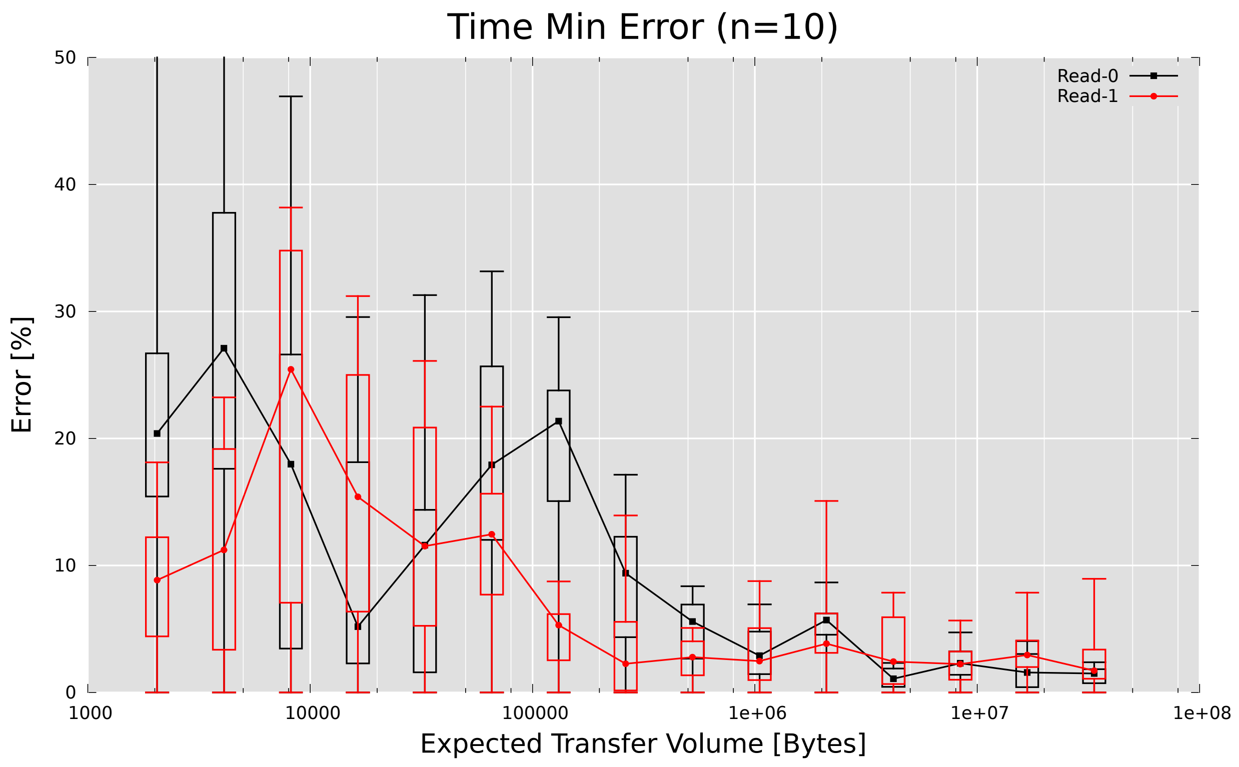


Figure C.31: Core: Error of the Execution Time of two Read Kernels running at the same time with $K = 10$: $\mathrm{err}\left(\frac{\text{Execution Time}_{10}}{\min(\text{Execution Time})}\right)$

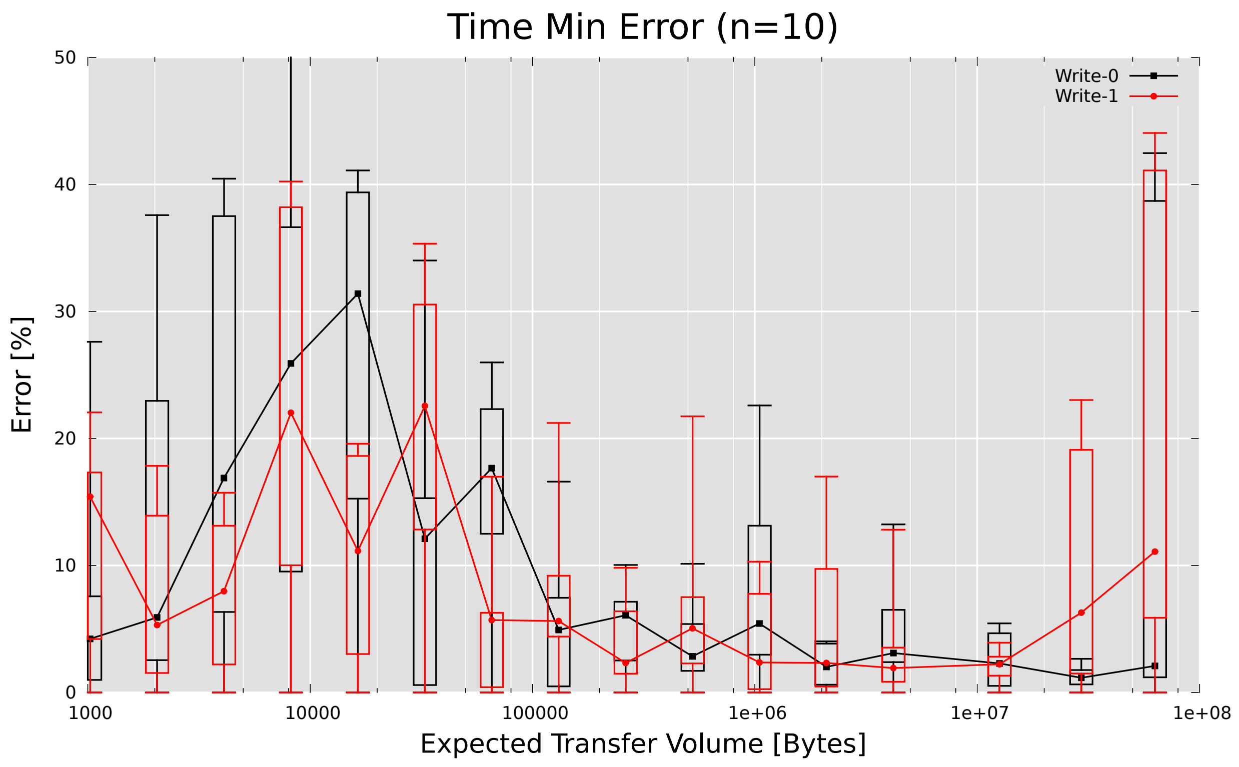


Figure C.32: Core: Error of the Execution Time of two Write Kernels running at the same time with $K = 10$: $\mathrm{err}\left(\frac{\text{Execution Time}_{10}}{\min(\text{Execution Time})}\right)$

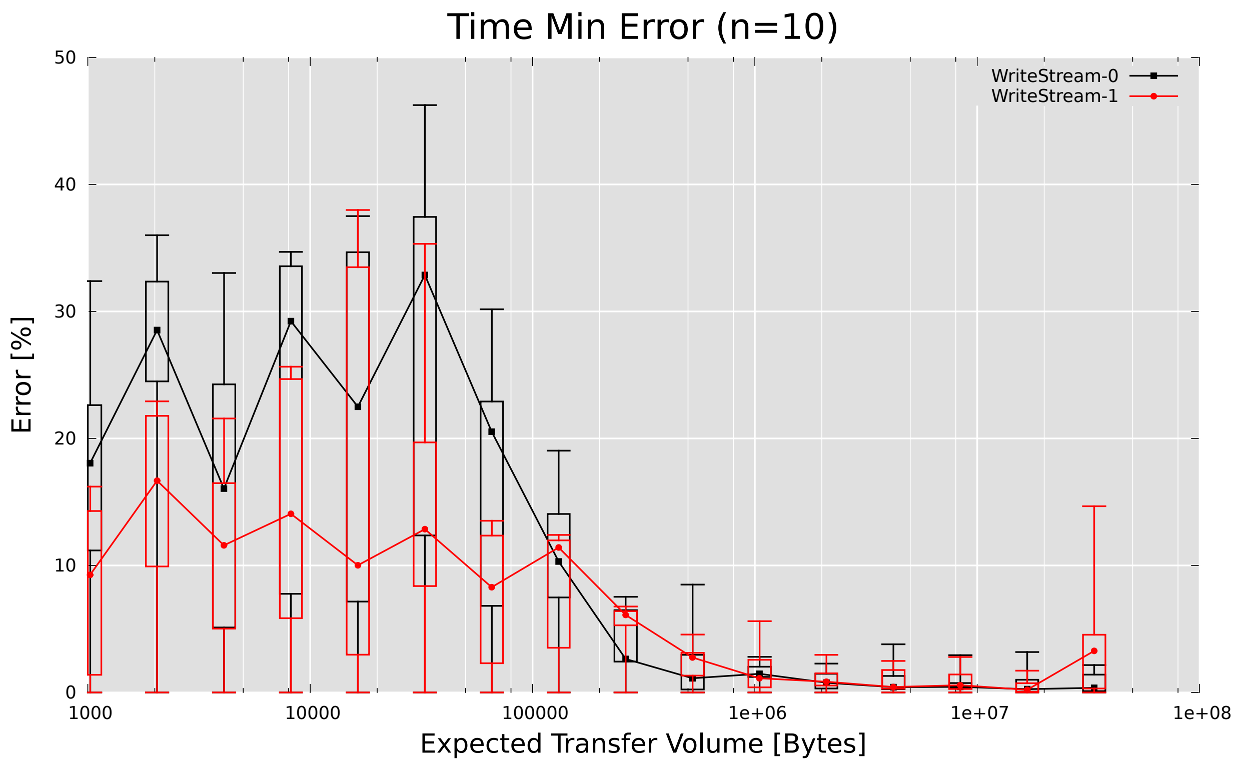


Figure C.33: Core: Error of the Execution Time of two Write Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

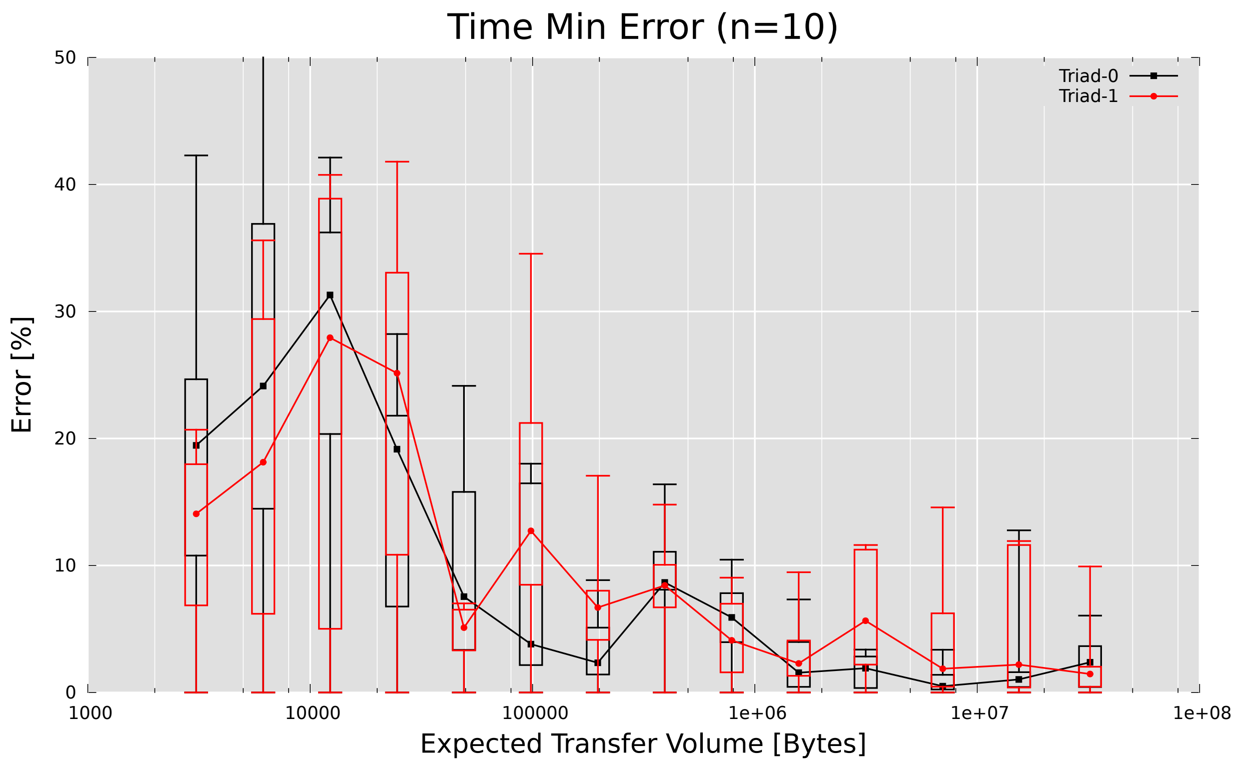


Figure C.34: Core: Error of the Execution Time of two Triad Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

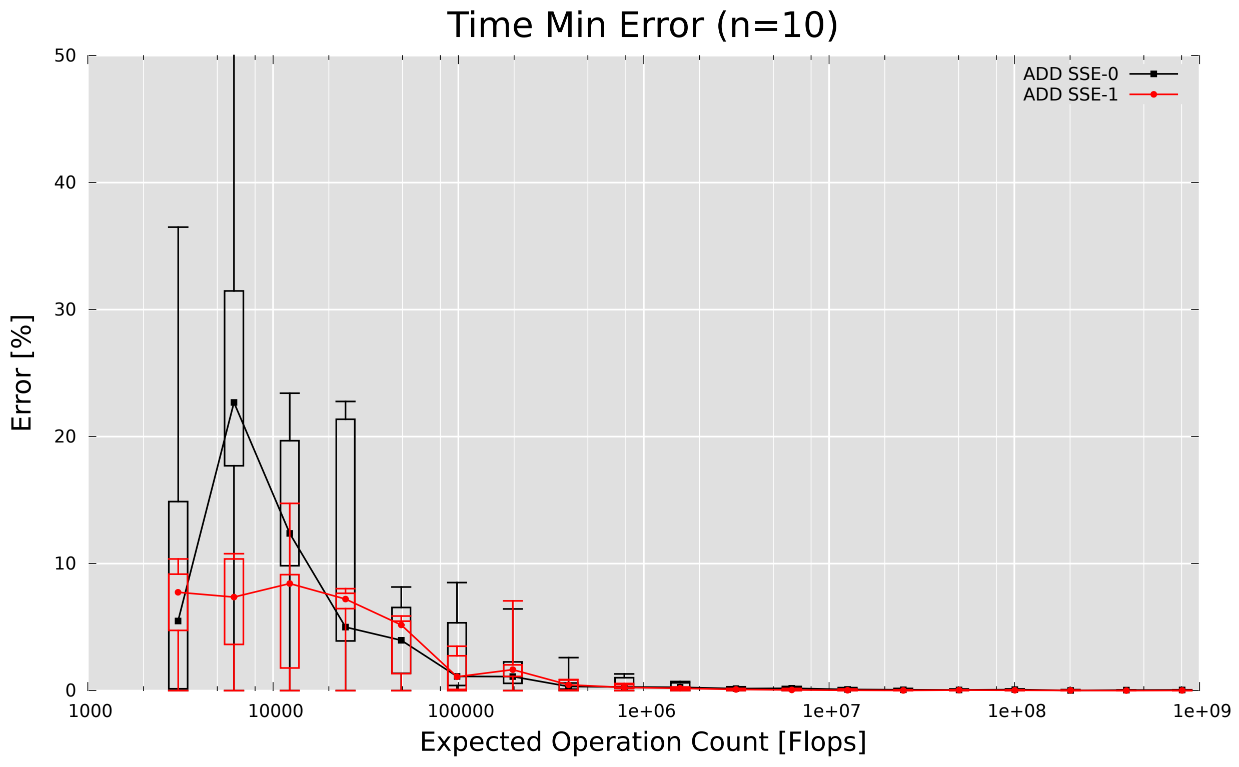


Figure C.35: Core: Error of the Execution Time of two ADD Kernels running at the same time with $K = 10$: $\text{err}\left(\frac{\text{Execution Time}_{10}}{\text{min(Execution Time)}}\right)$

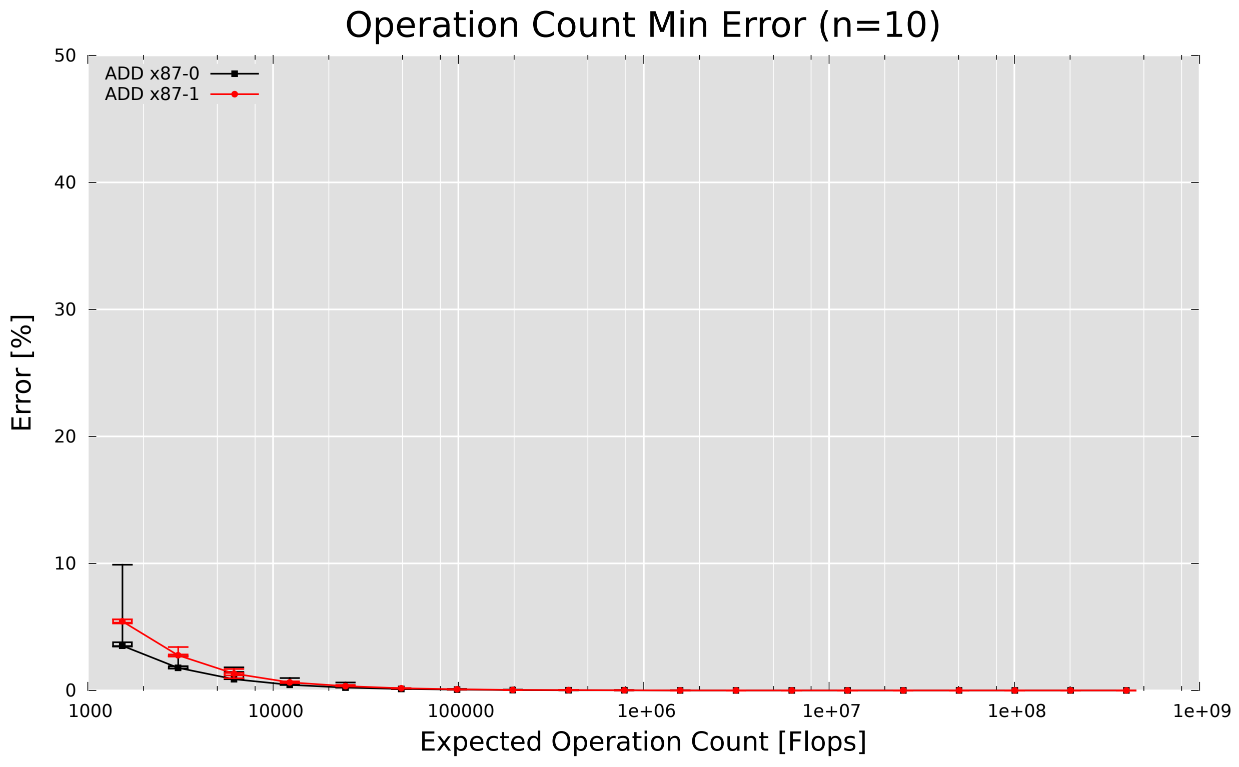


Figure C.36: Core: Error of the Operation Count of two ADD Kernels running at the same time with $K = 10$: $\text{err}(\frac{\text{Actual Operation Count}_{10}}{\text{Expected Operation Count}})$

C.4 Experimental Results

C.4.1 BLAS

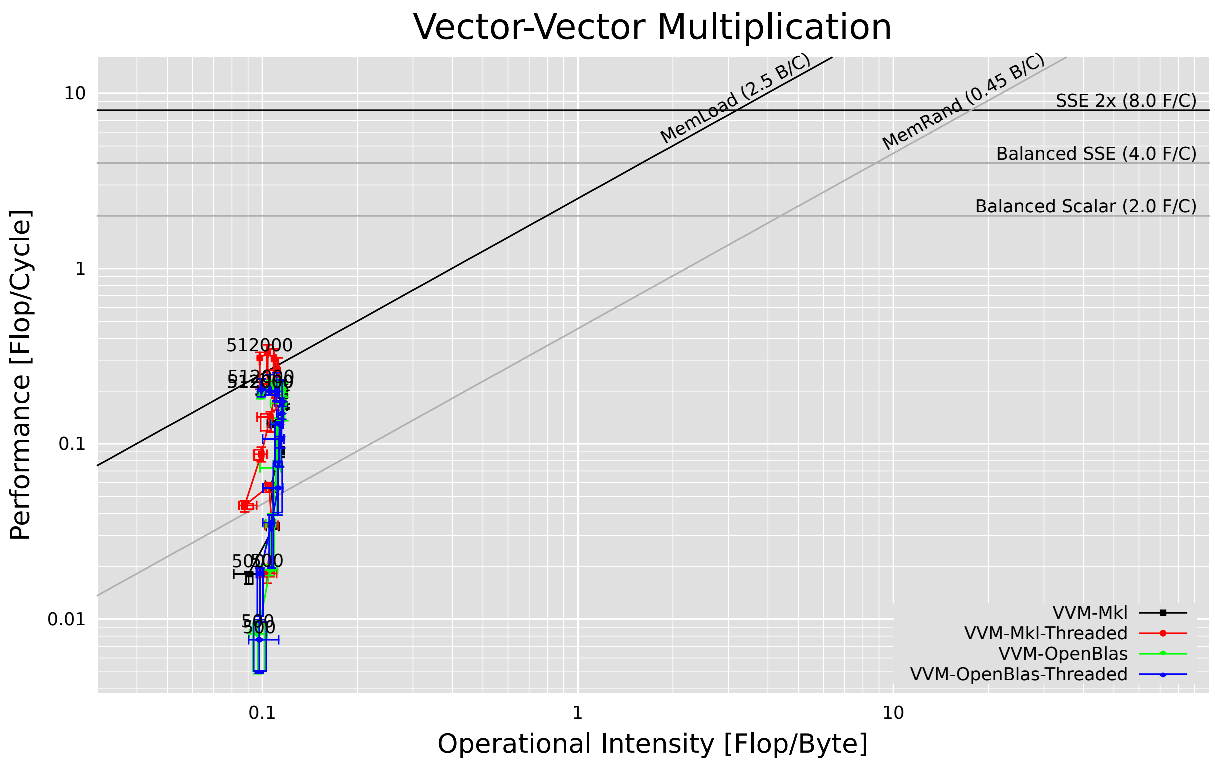


Figure C.37: Core: Roofline Plot of the Vector-Vector Multiplication (MemL2, DoublePrecisionFlop)

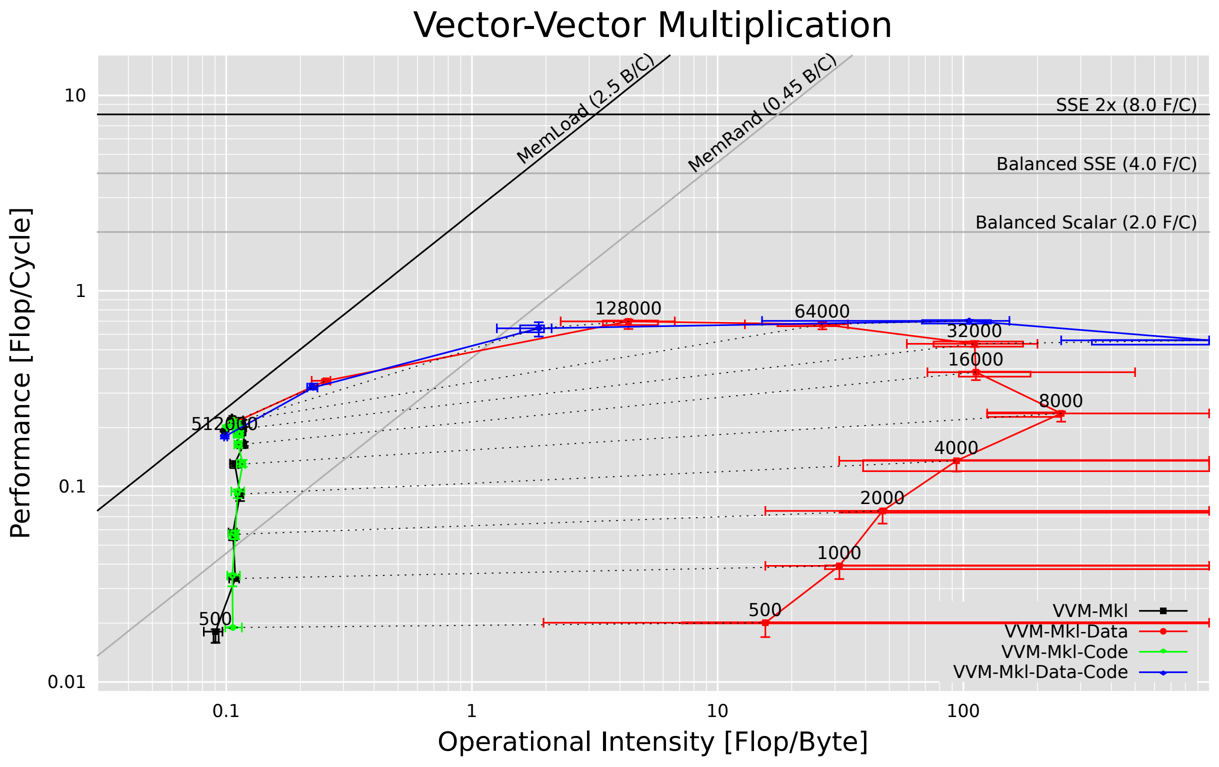


Figure C.38: Core: Roofline Plot of the influence of the initial cache state on the Vector-Vector Multiplication (MemL2, DoublePrecisionFlop)

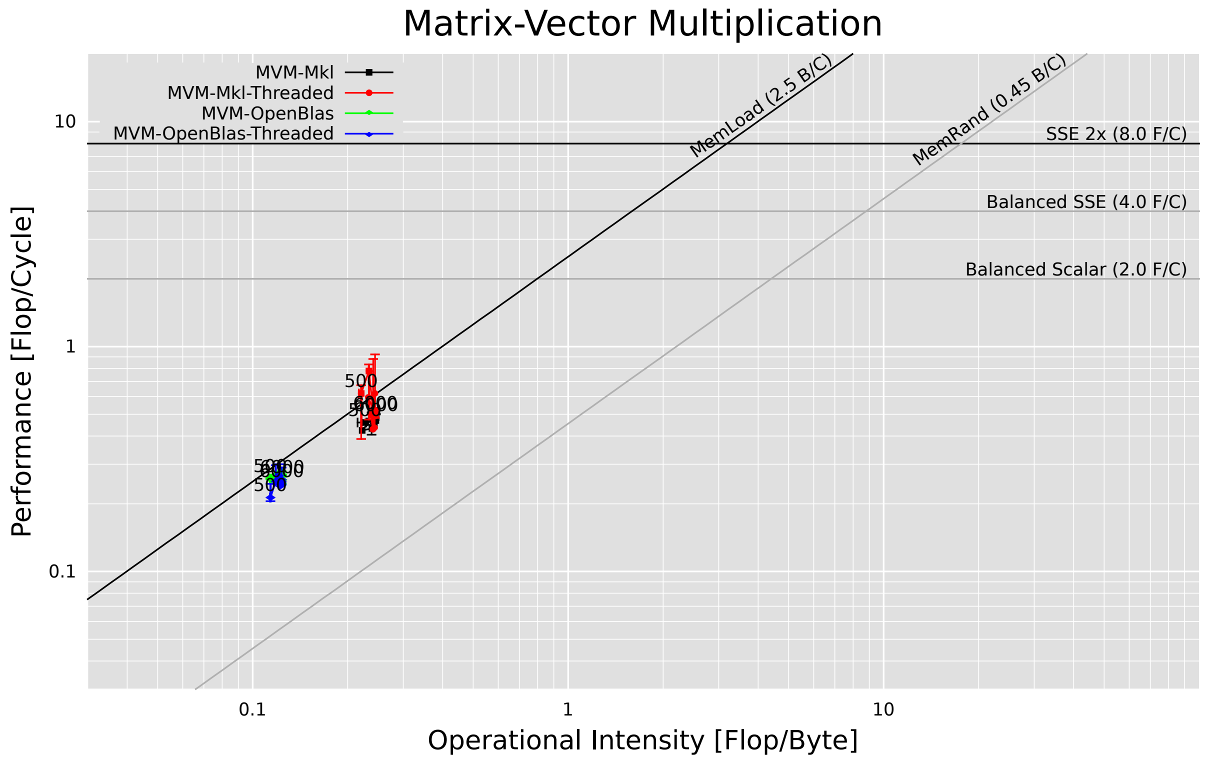


Figure C.39: Core: Roofline Plot of the Matrix-Vector Multiplication (MemL2, DoublePrecisionFlop)

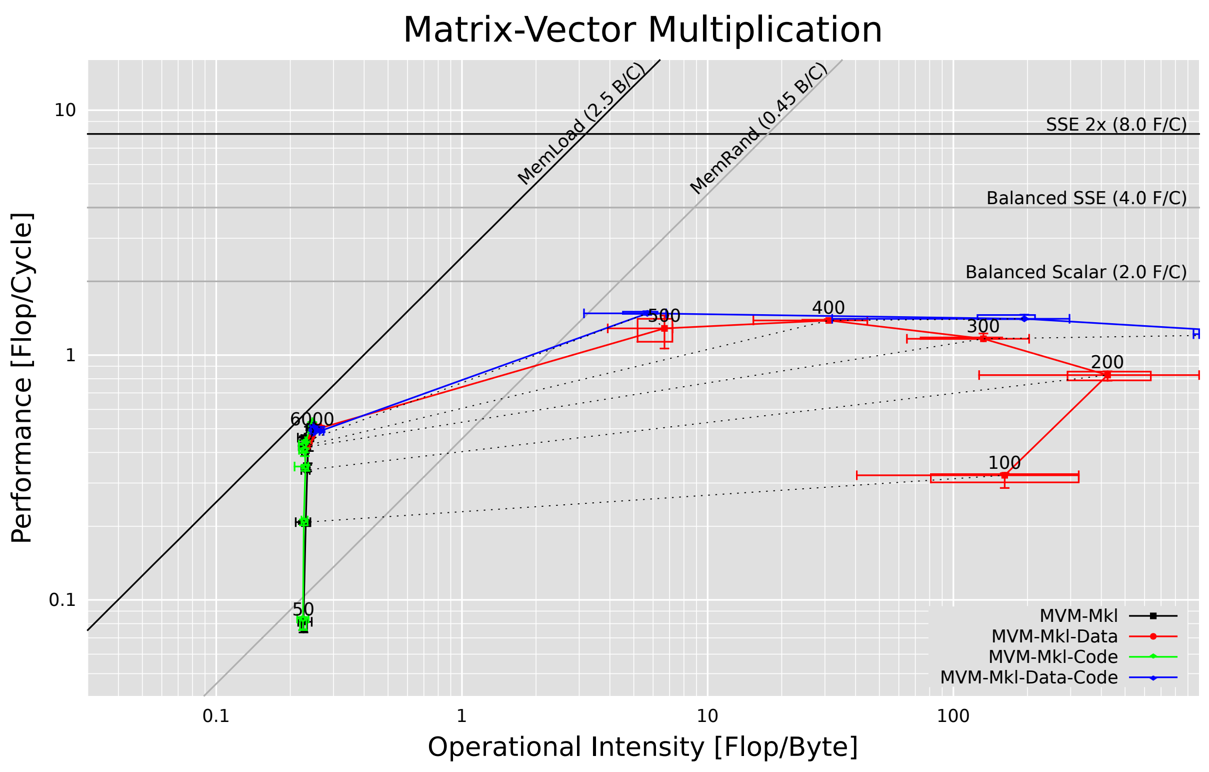


Figure C.40: Core: Roofline Plot of the Matrix-Vector Multiplication (MemL2, DoublePrecisionFlop)

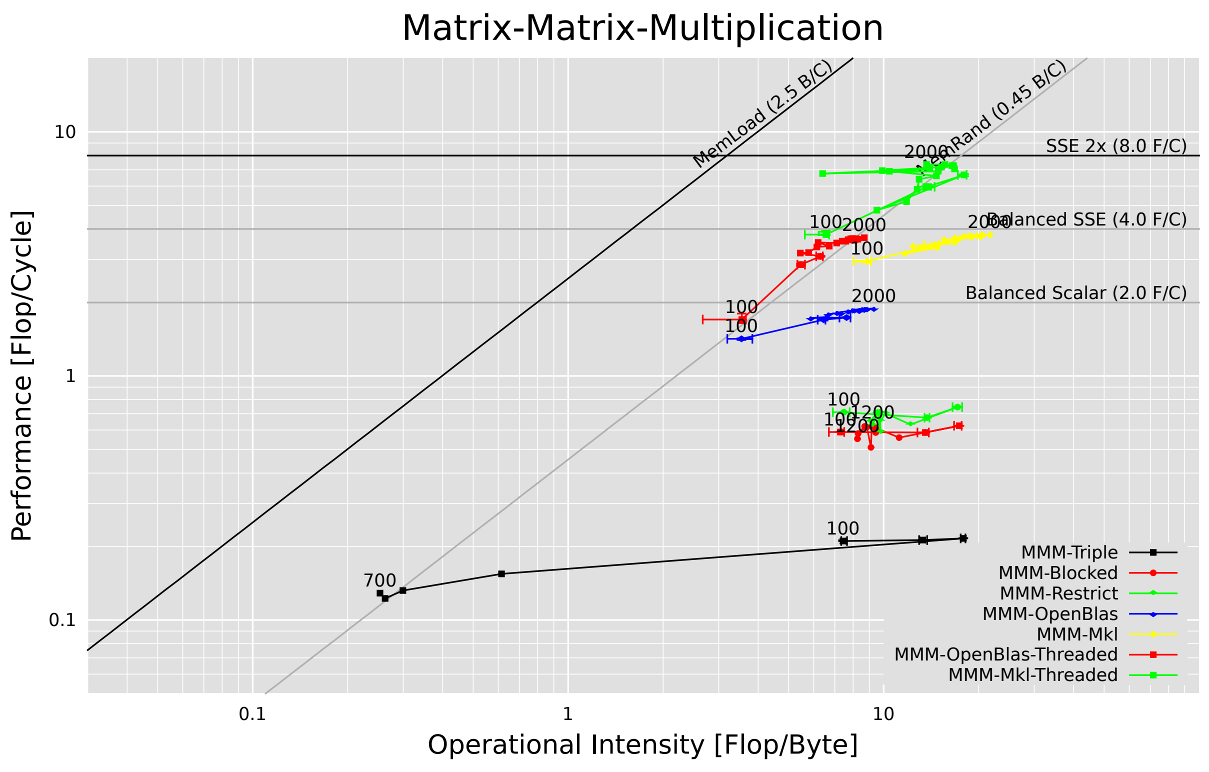


Figure C.41: Core: Roofline Plot of the Matrix-Matrix Multiplication (MemL2)

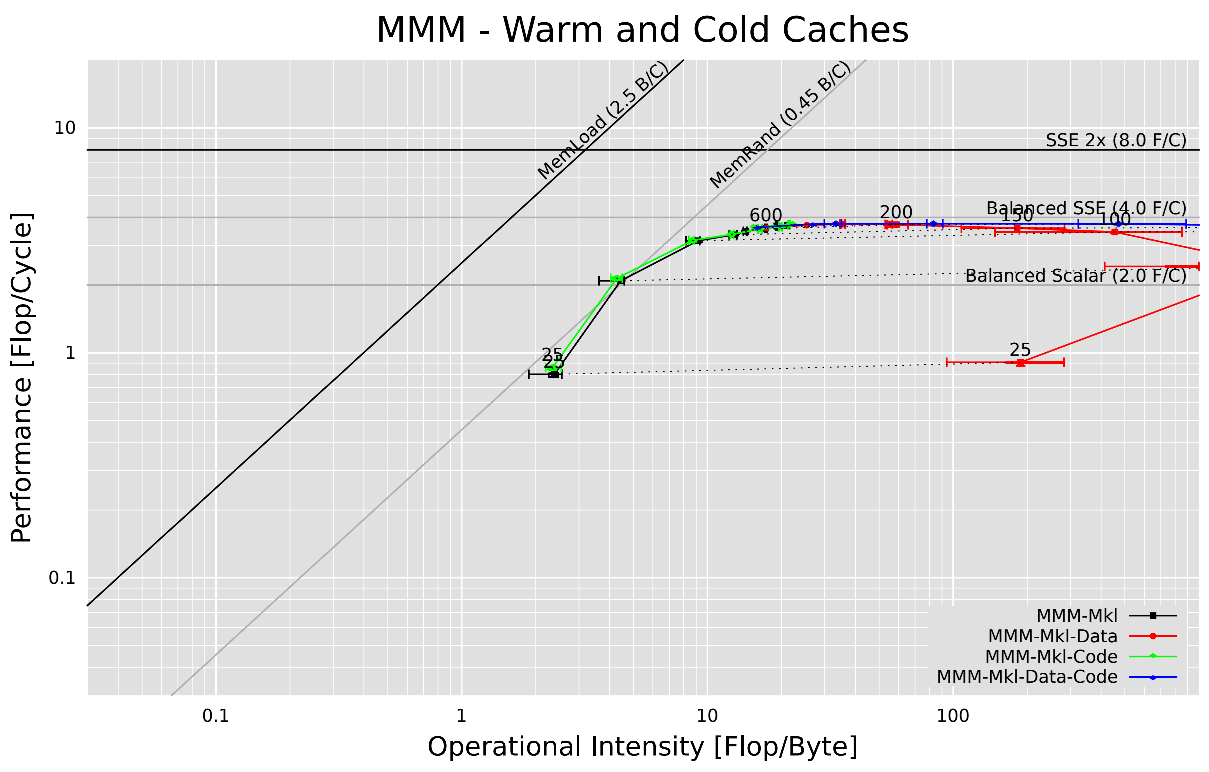


Figure C.42: Core: Roofline Plot of the Matrix-Matrix Multiplication with Different Initial Cache States (MemL2)

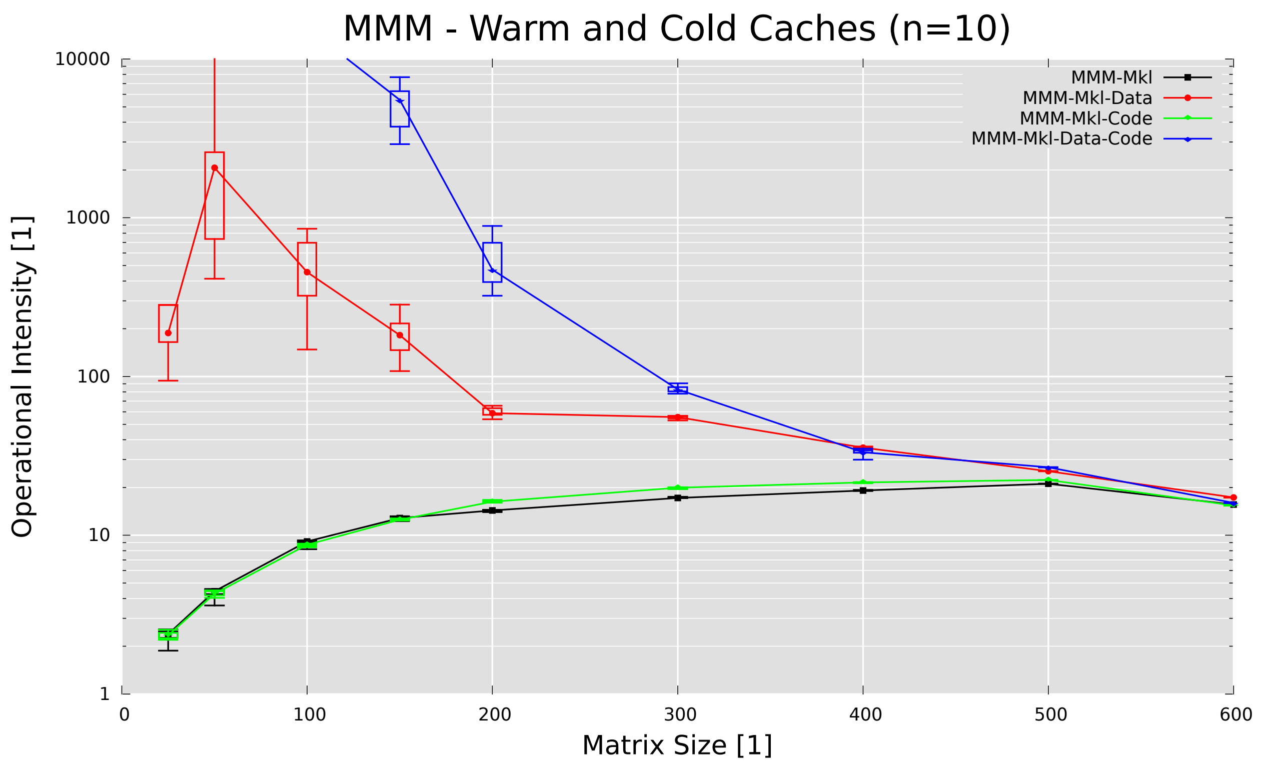


Figure C.43: Core: Effect of the Initial Cache State on the Operational Intensity for the Matrix-Matrix Multiplication (MemL2)

C.4.2 FFT

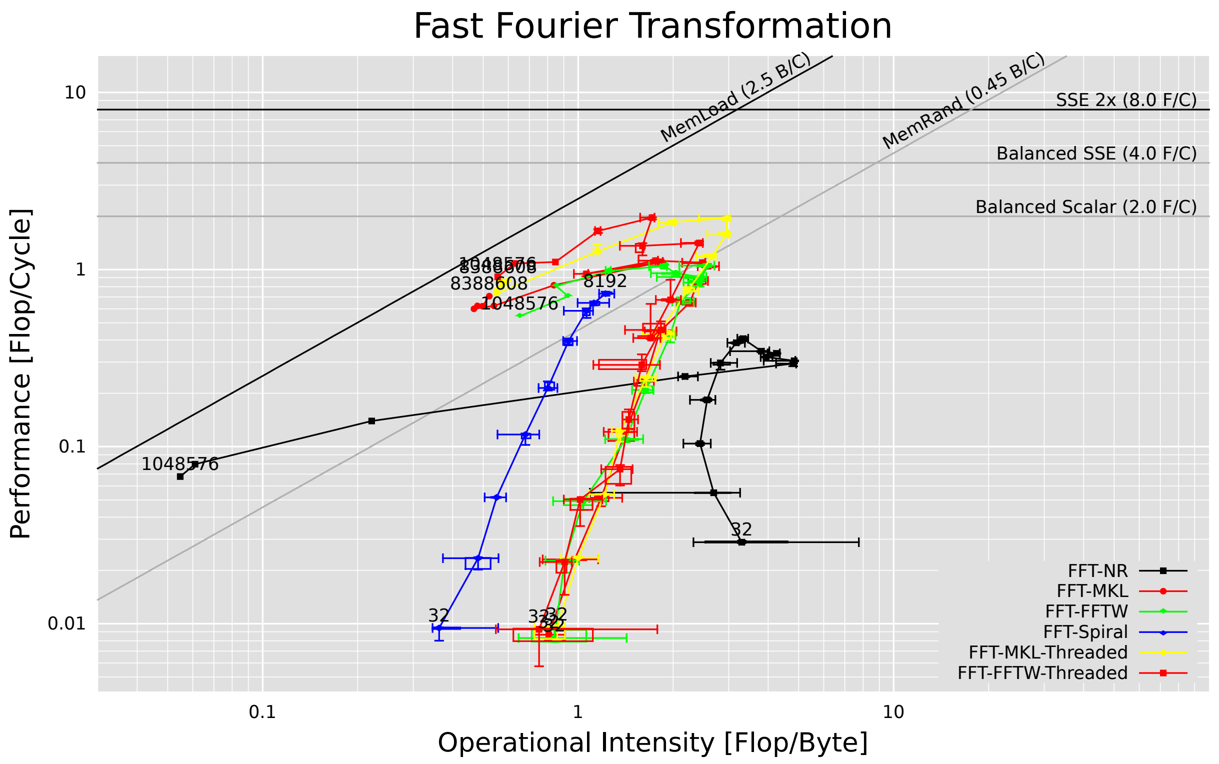


Figure C.44: Core: Roofline Plot of four FFT implementations (MemL2)

C.4.3 WHT

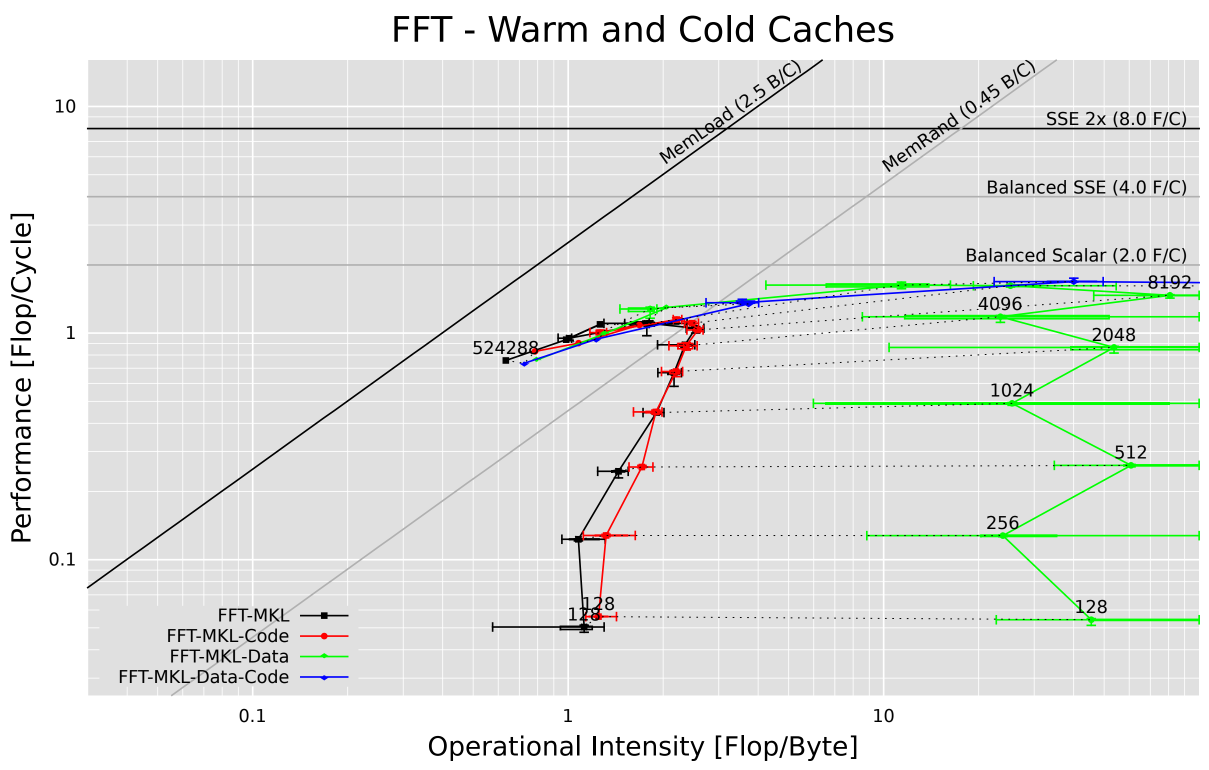


Figure C.45: Core: Influence of the initial cache state (MemL2)

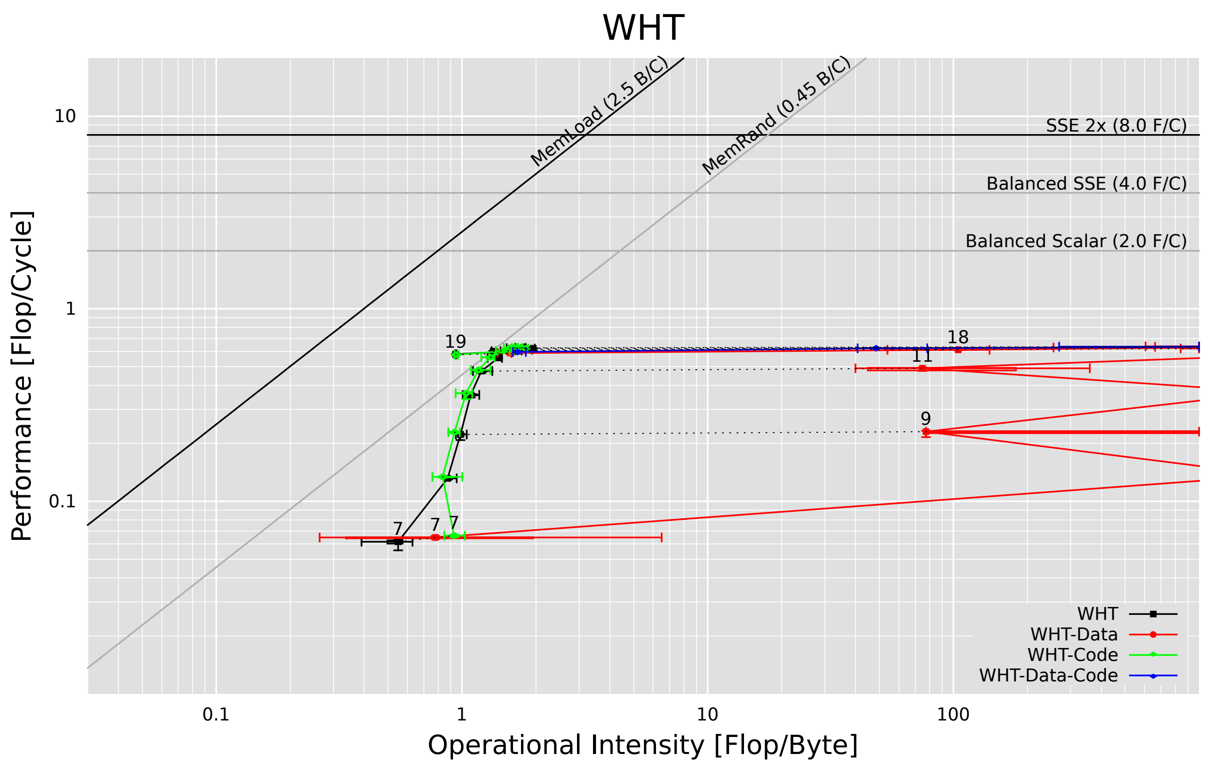


Figure C.46: Core: Roofline Plot WHT (MemL2)

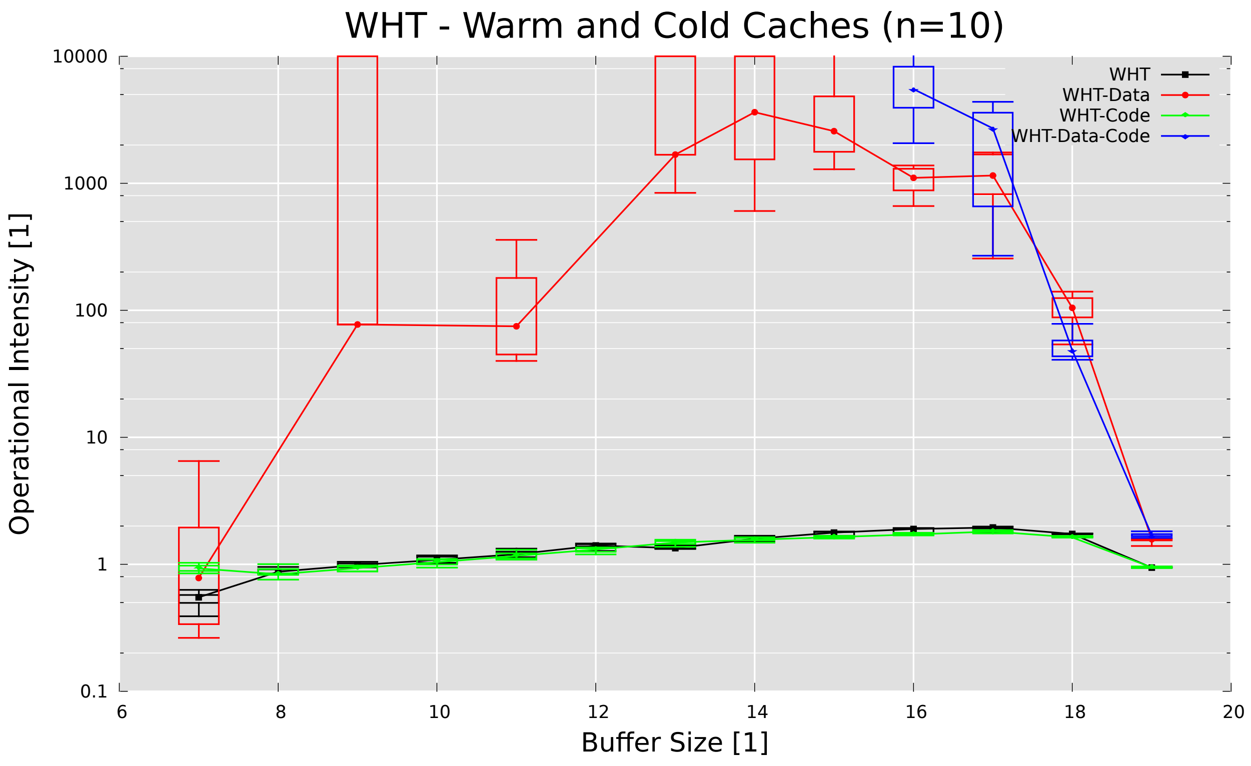


Figure C.47: Core: Operational Intensity WHT (MemL2)

List of Figures

Bibliography

[1] Fftw. http://www.fftw.org/. 55

[2] Libpfm4 documentation. http://perfmon2.sourceforge.net/docs_v4.html. 7

[3] Openblas. http://xianyi.github.com/OpenBLAS/. 47

[4] Spiral. http://www.spiral.net/. 55

[5] Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/. 45, 87

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. 5

[7] E. Boyd, W. Azeem, H.-H. Lee, T.-P. Shih, S.-H. Hung, and E. Davidson. A hierarchical approach to modeling and improving the performance of scientific applications on the ksr1. In *of Scientific Applications on the KSR1, Proceedings of the 1994 International Conference on Parallel Processing*, pages 188–192, 1994. 5

[8] R. Bryant and D. O'Hallaron. *Computer systems: a programmer's perspective.* Prentice Hall, 2011. 5

[9] E. Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley, 2004. 69

[10] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, Dec. 1989. 5

[11] Intel. Intel math kernel library. http://software.intel.com/en-us/articles/intel-mkl/. 47, 55

[12] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual, June 2011. 43, 86

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 3 edition, 2007. 55

[14] P. D. S. Richard M. Stallman, Roland McGrath. Gnu make version 3.82. Technical report, Free Software Foundation, July 2010. 81

[15] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005. 1

[16] V. Weaver. The unofficial linux perf events web-page. `http://web.eecs.utk.edu/~vweaver1/projects/perf-events/`. 7

[17] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Softw. Pract. Exper.*, 38(15):1621–1642, Dec. 2008. 5, 10

[18] Wikipedia. Accuracy and precision. `http://en.wikipedia.org/wiki/Accuracy_and_precision`. 11

[19] Wikipedia. Box plot. `http://en.wikipedia.org/wiki/Box_plot`. 13

[20] Wikipedia. Fast fourier transform. `http://en.wikipedia.org/wiki/Fast_Fourier_transform`. 55

[21] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008. 1