Compiler Comparisons using Performance Counters

Or the design of a framework for using Performance Counters to evaluate compiler logic and code generation.

Rune Erlend Jensen

CERN Openlab phase III
Submission Date: Sep 2009
Supervisor: Sverre Jarp

European Organization for Nuclear Research
IT Department

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report is intended to describe all the work performed during my internship at CERN. Because to the nature of the work, the results found and time constrains, this is only partially the case. It consists of three parts with the description of a framework developed as the main topic. In order to document as much as possible of the implementation and possibilities presented, several parts are not complete. There may be small errors which should be reported for correction.

Chapter 2

Technical Background and Concepts

In order to clarify the details of the implementations, key concepts are presented in this chapter.

2.1 Hardware Performance Counters

In order to debug and evaluate the inner workings of modern microprocessors, a method for collecting internal events can be critically important. In order to implement this, a set of internal logic and registers are employed, capable of detecting and acting on various events as they occur. This functionality is often refered to as *Hardware Performance Counters*. More detailed documentation can be found in *Intel® 64 and IA-32 Architectures Optimization Reference Manual*[1].

2.2 Theoretical Performance

When evaluating performance there are several measures one can use as a basis. A common one is *Cycles per instruction* (CPI), indicating the number of cycles it takes to perform each instruction in a program. By comparing the CPI with the theoretical capability of the processor, it is possible to see how well the application performs in an average cycle on the processor. A low number indicates that the code is being executed in an efficient way, and this is considered to be good. Thus, one goal of a hardware architect is to minimize the CPI of important programs like the test suites from *Standard Performance Evaluation Corporation* (SPEC).

For a software architect CPI can not be treated in the same way, that is, as a measure of good program efficiency. With software, the CPI can be manipulated or changed with ease. Inserting simple (and useless) null-operations into the code will rapidly improve the CPI, but execution time will increase as well. Therefore, using CPI as a guideline when developing software can be misleading. A better measure might be to use the time required for an operation to complete, but this will not give much indication of program efficiency.

In order to find a useful measure of the *maximum theoretical performance* of a program, several issues must be considered. In order to simplify the task, one can start by evaluating a single function. Then one must identify an important required basic operation; for math intensive code this will likely be either additions and/or multiplications. By counting the number of these key operations performed and dividing by the time used to execute the function, it is possible to measure the throughput of useful work. Equation 2.1 show how this might be calculated.

$$Work_{Useful} = \frac{ImportantRequiredOperationCount}{TimeSpent} \tag{2.1}$$

In order for Equation 2.1 to be more informative, it is useful to change the time into clock cycles spent by the processor. Calculating the number of cycles used by the CPU in an given interval is shown in Equation 2.2. By combining Equation 2.1 and 2.2 into Equation 2.3, we get a useful measure of throughput each cycle. Now it is possible to compare the CPU's theoretical throughput each cycle (of the selected operations) with the value from Equation 2.3, and obtain a good indication of how close to the theoretical maximum the evaluated code is.

$$CPU_{Work\_Cycles} = \frac{TimeSpent(sec)}{Processorfrequency} \tag{2.2}$$

$$Work_{Usefulpr.Cycle} = \frac{ImportantRequiredOperationCount}{CPU_{Work\_Cycles}} \tag{2.3}$$

2.3 Hardware Test Beds

The server *olbl0131* was used for all testing. It consists of two quad core Intel® Xeon® E5450 (Harpertown) processors running at 3 GHz. The details of the processors can be found in Table 2.1.

Table 2.1: olbl0131 processor details.

Cpu family	6
Model	23
Model name	Intel® Xeon® CPU, E5450 @ 3.00GHz
Stepping	6
L1 cache size	32 KB
L2 cache size	6144 KB

Chapter 3

Framework for Performance Evaluation

A common problem when trying to optimize code is knowing when a change leads to any improvement. While large performance gains are easy to detect, they are also infrequent. Most of the improvements are small, but they can add up to become useful. In order to have rapid, detailed and very accurate feedback, a framework using hardware performance counters has been made. The functionality, design and usage of a prototype version is described, along with insight into usage of performance counters.

3.1 The Basic Idea

A number of benchmark programs are compiled numerous times. Each program is compiled with a selection of compilers and compiler flags. Thus, for each compiler, several flag combinations are tested. This process is also performed for different data sizes of vectors and matrices.

After a benchmark program is compiled, it is executed several times using pfmon[1] to gather stats from each runs. The statistics are then used to identify various performance issues and to calculate near cycle-exact timings.

[1] `http://perfmon2.sourceforge.net/`

3.2 Possibilities

Two possibilities exist. The first is to evaluate a single build of a program, when one performs optimizations and/or testing. The other is with a broad scan over multiple builds and programs, compilers, compiler flags and data sizes.

A key feature is the use of graphs to illustrate how performance changes with data size, comparing several different performance counter events, and using it to find patters and connections. This allows fast(er) understanding of how each event works, and whether an event is a cause or side effect. As such, this framework can also function as an educational tool.

The primary intended usage during this internship, however, is to evaluate performance of several SMatrix/SVector functions using various compilers.

The second intended usage is to find out which compiler flags produce the most efficient code.

The third intended usage: Developers - find both where to optimize and helping to test new code versions/algorithms.

The fourth intended usage: Give users an idea of the performance cost of functions.

The fifth intended usage: Give compiler developers rapid feedback (performance, artifacts, regressions and bugs).

The sixth intended usage: Enable performance predictions for users, before migrating to a new compiler/library version/CPU.

3.3 Implementation

Two sets of measurements are obtained for each program, where only the number of loop iterations performed by the program are different. Thus, the program reads a loop count as a parameter when being executed, in order enable different measurement sets. This allows computation of the cost of a single loop iteration, eliminating any constant overhead generated by loading, setup and start-up branch mispredictions. This requires that the tested program does not perform any secondary activity related to the number of iterations specified. In particular, the size and/or initialization of data structures must not depend on the loop count - unless they are the target for analysis. If there are initialization(s) that changes based on loop count, a large default size must be initialized (at least as large as the largest loop count requested by the framework). This minimum size will then appear as a fixed setup cost, and will not affect the cost calculation of the loop iteration. With more complex interactions between loop count and initialization it might be required to first initialize different structures for all loop counts used by the framework, and then select the right one based on requested loop count (keeping the work in the initialization phase constant).

Currently, the first measurement point selected is 1 (which should be multiplied by 100 by the program). The second measurement point selected is dynamically selected based on the program runtime, and will be either 10000, 1000 or 100. This is performed in order to avoid unnecessarily long execution time for more complex programs. Limited evaluation of the effect of reducing the loop count indicates that the quality is not reduced, at least for programs with somewhat longer execution time. The formula used to obtain the number of events for a single loop iteration is shown in Equation 3.1. More detailed information on the inner design of the test programs used can be found in Section 3.3.1.

$$Count_{loop} = \frac{eventCount\_100xx - eventCount\_1}{runCount100xx - 1} \tag{3.1}$$

Multiple performance counter events are used and in the current version 51 different ones are collected. In order to get precise counts neither multiplexing or statistical sampling is used, only basic counting of a single process. This requires multiple repeated runs, each with a single set of events.

Each run will normally gather measurements of different "precise events". Precise events in this context are not necessarily PEBS[2] based, but events with small variations between repeated runs of the same program. Some examples are BRANCH_INSTRUCTIONS_RETIRED, SIMD_COMP_INST_RETIRED:PACKED_DOUBLE and SIMD_INST_RETIRED:ANY.

[2]Precise Event Based Sampling

Each run can also gather extra samples of unstable events. These are events that show larger fluctuations between runs. Currently this is only performed for selected events like RAT_STALLS:ANY, RESOURCE_STALLS:ANY, RESOURCE_STALLS:LD_ST and RS_UOPS_DISPATCHED_NONE. Several other events belong in this category, but are only sampled once. The main reason is to reduce the total execution time, sacrificing accuracy for less informative events. When more information is needed, or if fluctuations seem too high, it might be useful to add more samples of that type of events. Events types like RAT_STALLS and RESOURCE_STALLS are likely candidates for this.

Every run also gathers extra samples of the most unstable events. Currently this is only performed for UNHALTED_CORE_CYCLES and UNHALTED_REFERENCE_CYCLES. Both of these events have dedicated counters, so the cost of including them for each rerun is minimal. Since the event INSTRUCTIONS_RETIRED also has a dedicated counter, it is also sampled in every run, even when it is very stable.

3.3.1 Benchmark Program Design

When creating small programs for performance evaluation, a constant problem is to avoid having the compiler optimize away too much. Often, several parts of the code can be identified as "useless" or invariant. This leads to benchmarks with little or no value. Having a section of code (like a loop) without any of the answers used might therefore be removed by the compiler, except when optimization is turned off. To avoid this behavior it is possible to print parts of the answer, or return it at the end of the program. Unfortunately, this is not always enough.

If only parts of the calculation are needed to create the output, only those parts might be actually performed. This works in several ways. First, consider a loop that performs floating-point vector-vector addition. After the loop is completed one prints a single value of the answer vector. This means that only one specific index of the resulting sum needs to be computed, the other parts might be removed. Too avoid this, one might sum up all the values in the vector after the loop, touching every element. Second, if the answer (or part of it) in every loop iteration is the same, it might be performed only once. One way to avoid this is that every loop iteration should work on different input values. Using large arrays of random data, with a different (small) part every iteration is possible, but might create cache related performance artifacts. Third, reusing the values produced in the previous iteration might solve the second issue, but might introduce a new ones. A problem is that this reuse might lead to over/under-flows or Inf/NaN's, which may affect performance. Also, with vector-vector addition it is possible to rewrite the calculation, replacing the looped additions with a single multiplication. This removes the loop, gives the correct answer (possibly with less round-off error), and makes the benchmark useless. Fourth, using zeros (0 or 0.0) in math calculations will avoid overflows, but are also excellent targets for removal. To some extent this is also true for 1.0 and any other constants, depending on the context.

The extent of compiler code removal depends on several factors, but two factors play a critical role. As the complexity of the code goes up, tracking of data dependencies is lost. This means that complex benchmarks suffer less from the problems mentioned above, as the repeated work is not detected. A different factor is the vendor and version of the compiler. Some vendors have better and more extensive analytical capabilities, leading to potentially big performance differences in benchmarks. Typically, newer compilers also have better analytical capabilities, leading to problems where old benchmarks no longer behave as intended. To some extent optimization flags will modify this behavior, but the common -O2 will typically turn on all removal features. Using either -O0 or -O1 might prevent code removal, but the usefulness of benchmarks is lost (unless the final program is compiled with the same flags).

In order to create a future-proof, side-effect free, low overhead, stable and fair way of creating benchmark code, several designs have been tested. To prevent code from being removed in low level drivers, operating systems and libraries the use of *volatile* data is common. This instructs the compiler to assume that the associated data is modified by external means. By using volatile, it becomes possible to force the compiler to perform specific tasks. Program 1 shows this technique. Every loop iteration requires loading the variable *zero_T*, and writing it to the vector *vec*. The index *zero* is also read every iteration. Both loads must be performed and the compiler must assume they have different values every time. This means that it is not possible to remove any of the computations. Note that the data size given by *#define DIM_L 5*, reflecting the data size used in the graphs, is updated as needed in the framework (being constant at compile time).

Program 1 Usage of volatile to modify an SMatrix array. First attempt.

```
#define DIM_L 5
volatile int zero = 0;
volatile double zero_T = 0;
volatile double one_T = 1;

SVector<double, DIM_L> vec;
SVector<double, DIM_L> vect;

vec[zero] = one_T;
vect[zero] = one_T;

for(int i = 0; i < 100*runs; ++i)
{
  vec[zero] = zero_T; // Force compiler to not remove any code
  vec += vect;
}
outputAcc += vec[zero];
return static_cast<int>(outputAcc);
```

While the code in Program 1 behaves theoretically correctly (unless carefully evaluated), there are some problems. With the Core 2 processor the write address will be correctly predicted (it is effectively constant), and the data can be forwarded directly into the following load. Unfortunately, it leads to performance issues with packed instructions as shown in Figure 3.1. A read from *vec[0]* creates a stall if and only if it is a packed one, giving a penalty to code with packed instructions, because forwarding can not be performed when the the size of data being read (two doubles) is larger that the written data (a single double) see in Figure 3.2. The problem is made worse because the write and following read most likely are very close to each other.

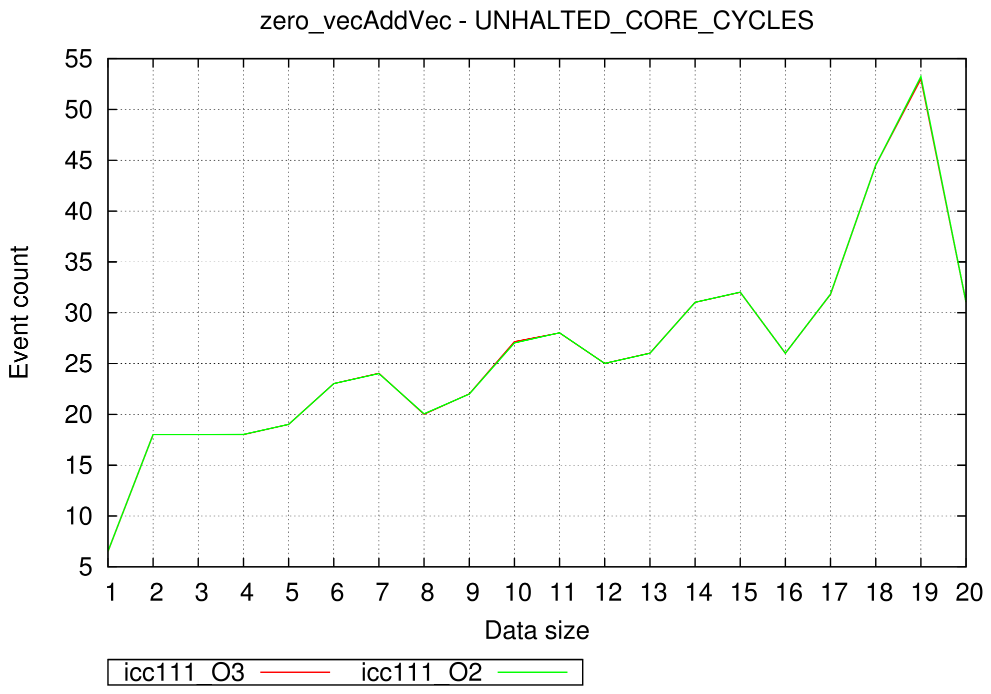

Figure 3.1: Cycle count for a single loop iteration of vector-vector addition in Program 1.

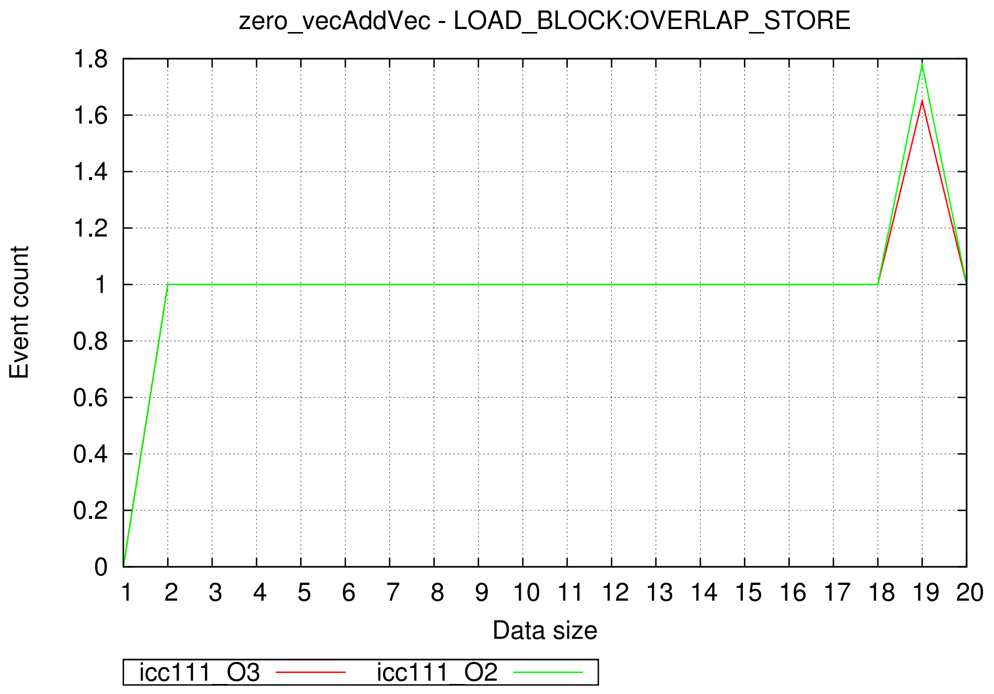

Figure 3.2: Partial data forwarding stall caused by a scalar write into the first element in Program 1.

A somewhat better method can be shown in Program 2 where the write is performed into the last element of the array. This increases the time between the write and the load, allowing the read to be performed only from the cache without any partial forwarding. The new performance can be shown in Figure 3.3, and the associated forwarding stalls in Figure 3.4. Now the forwarding only affects even data sizes, with the odd sizes using a scalar load for the last element.

Program 2 Usage of volatile to modify an SMatrix array. Second attempt.

```
#define DIM_L 5
volatile int zero = 0;
volatile int DIM_L_minus_1 = DIM_L-1;
volatile double zero_T = 0;
volatile double one_T = 1;

SVector<double, DIM_L> vec;
SVector<double, DIM_L> vect;

vec[zero] = one_T;
vect[zero] = one_T;

for(int i = 0; i < 100*runs; ++i)
{
  vec[DIM_L_minus_1] = one_T; // Force compiler to not remove
      any code
  vec += vect;
}
outputAcc += vec[zero];
return static_cast<int>(outputAcc);
```

This solution is not perfect as can be seen by the uneven performance graph in Figure 3.3. Several tricks have been tried in order to fix this problem. Third attempt to force code generation, with minimal impact, is shown in Program 3. This code creates a piece of inline assembly that can not be moved or removed. It also uses the *vec* array as output, telling the compiler it might change. Finally, it marks all memory modified by the inline assembly. The inline assembly is just a comment so it will not have any effect at all. Combined, this forces the compiler to reload and recalculate everything in each loop. The new performance is shown in Figure 3.5. For the small data sizes the performance gain is significant, and Figure 3.6 confirms that the forwarding issue is totally gone.

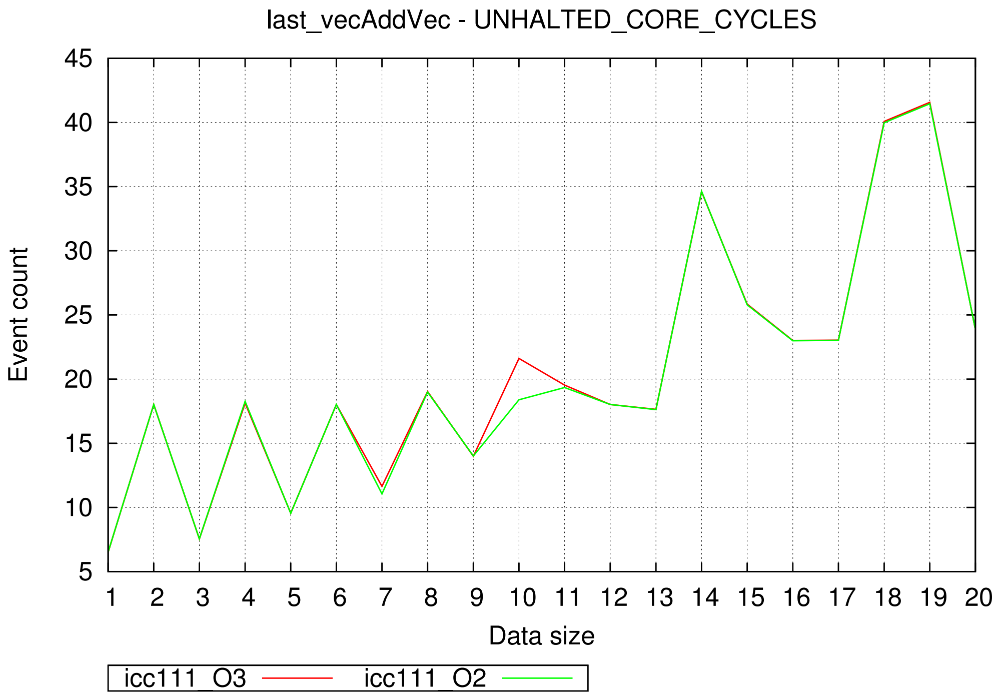

Figure 3.3: Performance when writing to the last array element every iteration.

Program 3 Usage of volatile to modify an SMatrix array. Third attempt.

```
#define DIM_L 5
volatile int zero = 0;
volatile double one_T = 1;

SVector<double, DIM_L> vec;
SVector<double, DIM_L> vect;

vec[zero] = one_T;
vect[zero] = one_T;

for(int i = 0; i < 100*runs; ++i)
{
  __asm__ __volatile__ ("# Nothing. Just trick the compiler."
    : "=o"(*vec.Array()) // Now the data in vec is marked as
       output.
    :
    : "memory"); // All the data (in vec) might change
       unpredictably, so reload.
  vec += vect;
}
outputAcc += vec[zero];
return static_cast<int>(outputAcc);
```

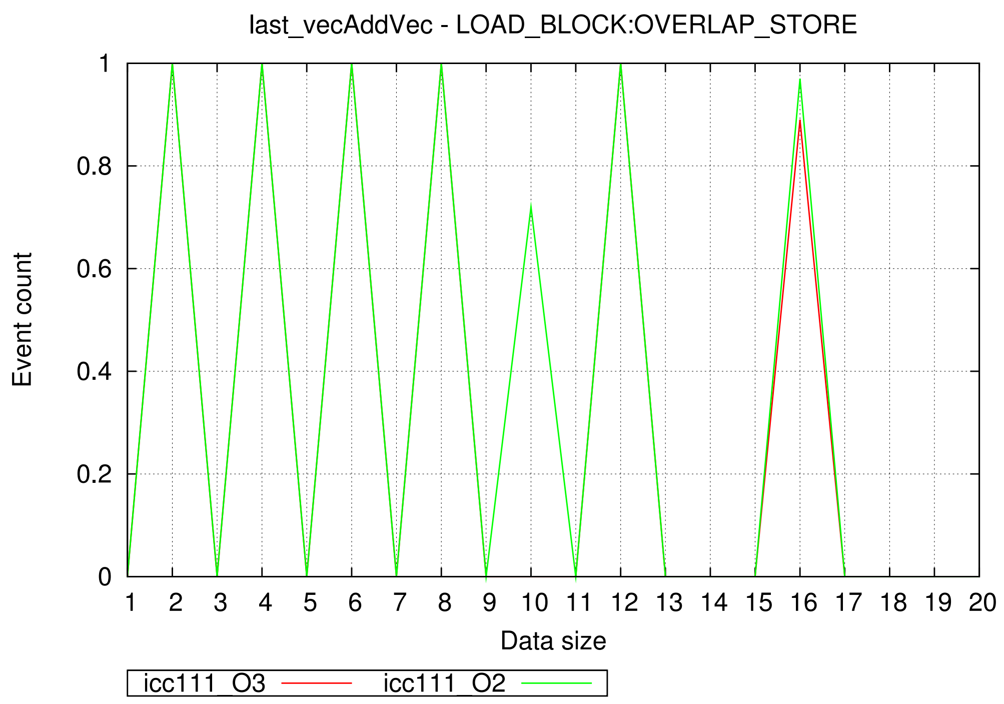

Figure 3.4: Partial data forwarding stalls, only created when the last array element is read by a packed instruction.

While the third method works, the reason was found to be different than intended. *All* data was marked as modified, not only the listed output. This leads to both arrays being reloaded from memory (Figure 3.7) every iteration.

The fourth way to force code generation with minimal impact is shown in Program 4. Here only the data in *vec* is marked as being changed in external ways. This enables the compiler to retain the data in *vect* in registers every loop iteration. As long as the size of the data structure listed is known by the compiler, this will work correctly. Also, if the compiler finds that *vect* is all zero, it might eliminate the addition totally (as it will have no effect). By using the same method on *vect* before the loop any static analysis like this is prevented. This method is the one utilized in the latest evaluation.

Figure 3.8 shows the number of load operations performed with the final volatile design. For size 1-5, 8-9 and 16-17 the number of loads are halved compared to the old number in Figure 3.7; however this is not the case for the other data sizes. The cause of this is the use of branches (see Section 3.5.1), and those will prevent *vect* from remaining in registers - forcing a reload.

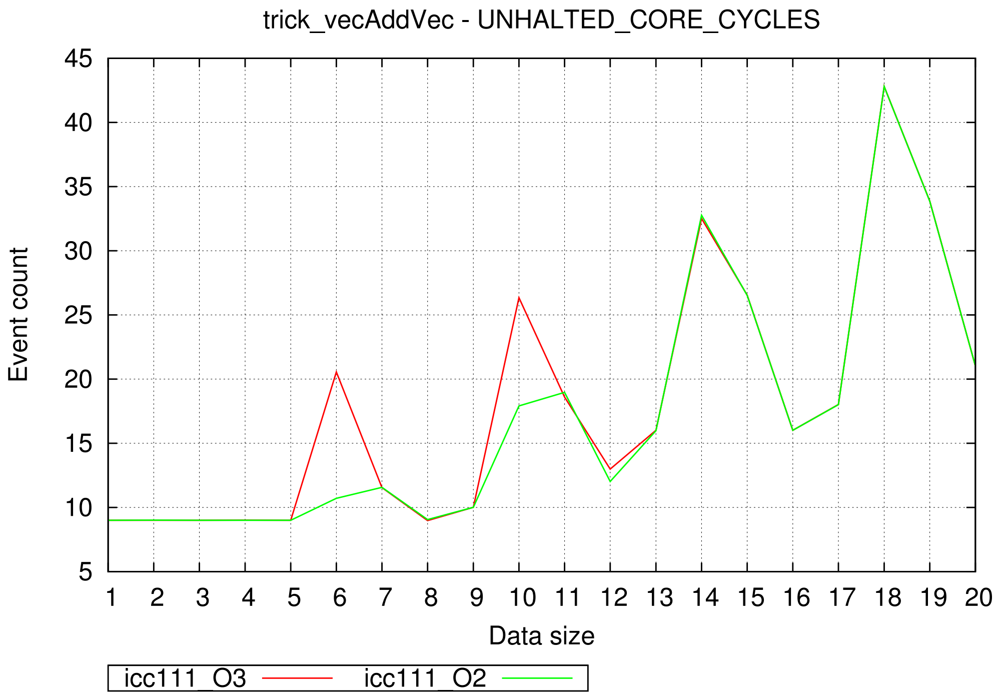

Figure 3.5: Performance of the third volatile design.

Program 4 Usage of volatile to modify an SMatrix array. Fourth attempt.

```
#define DIM_L 5
volatile int zero = 0;
volatile double one_T = 1;

SVector<double, DIM_L> vec;
SVector<double, DIM_L> vect;

vec[zero] = one_T;
vect[zero] = one_T;

// Make sure that vect is considered to contain unknown (non
    zero) values.
__asm__ __volatile__ ("# Nothing. Just trick the compiler."
     : "=o"(vect) // Now the data in vect is marked as output.
     : "o"(vect)  // Now the data in vect is marked as input.
     : );

for(int i = 0; i < 100*runs; ++i)
{
  __asm__ __volatile__ ("# Nothing. Just trick the compiler."
     : "=o"(vec) // Now the data in vec is marked as output.
     : "o"(vec)  // Now the data in vec is marked as input.
     : );
  vec += vect;
}
outputAcc += vec[zero];
return static_cast<int>(outputAcc);
```

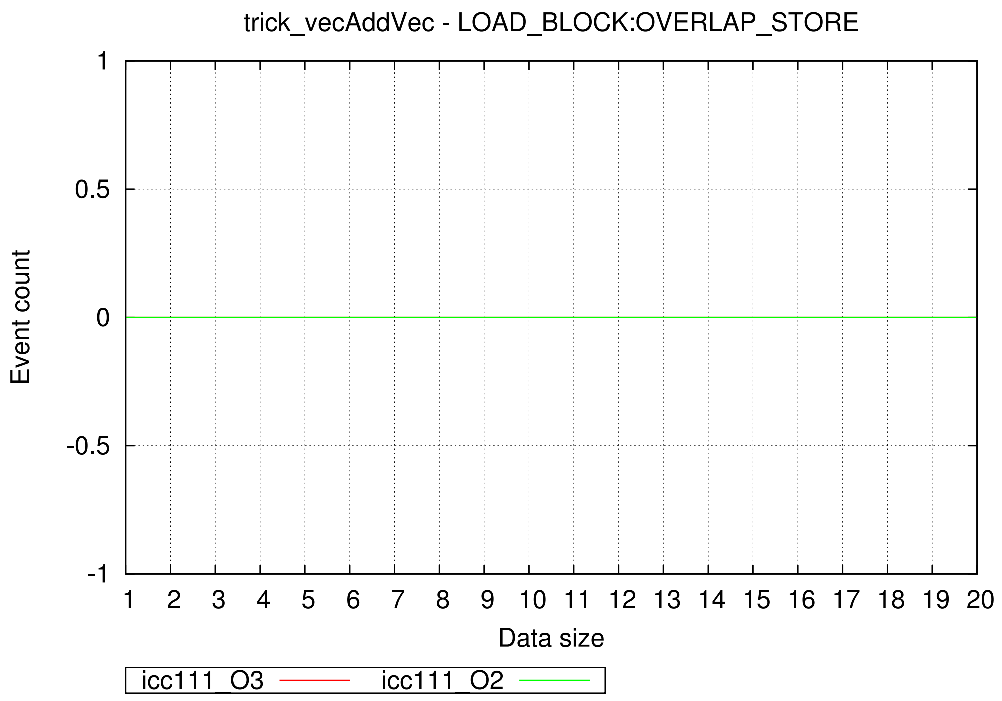

Figure 3.6: Partial data forwarding stalls are not created as all overlapping loads and stores use the same size.

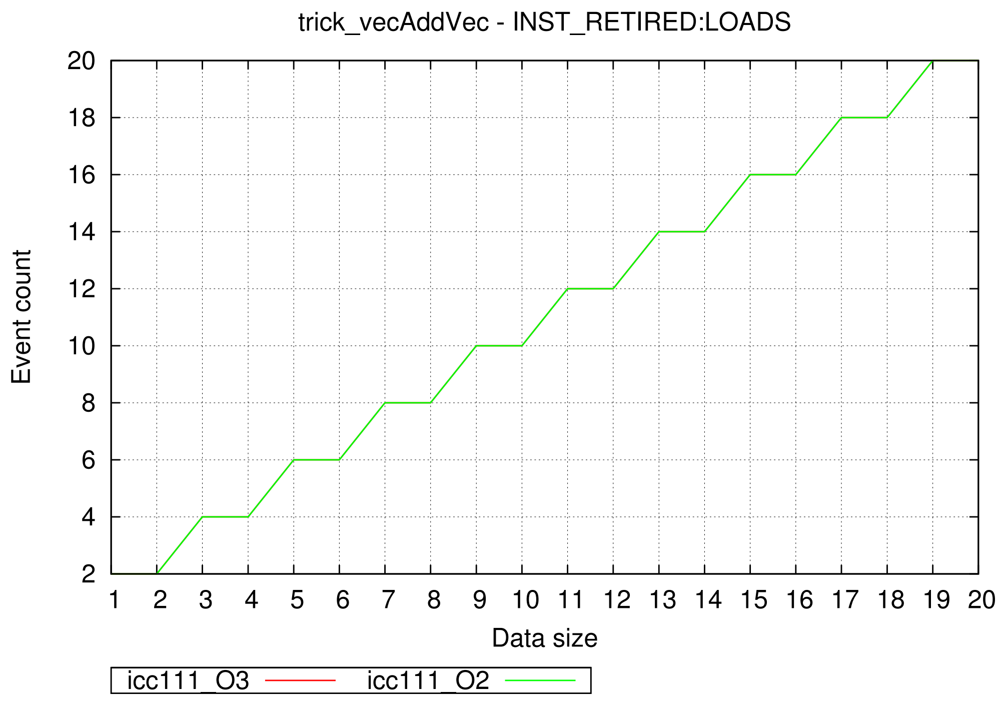

Figure 3.7: Number of loads performed with the third volatile design.

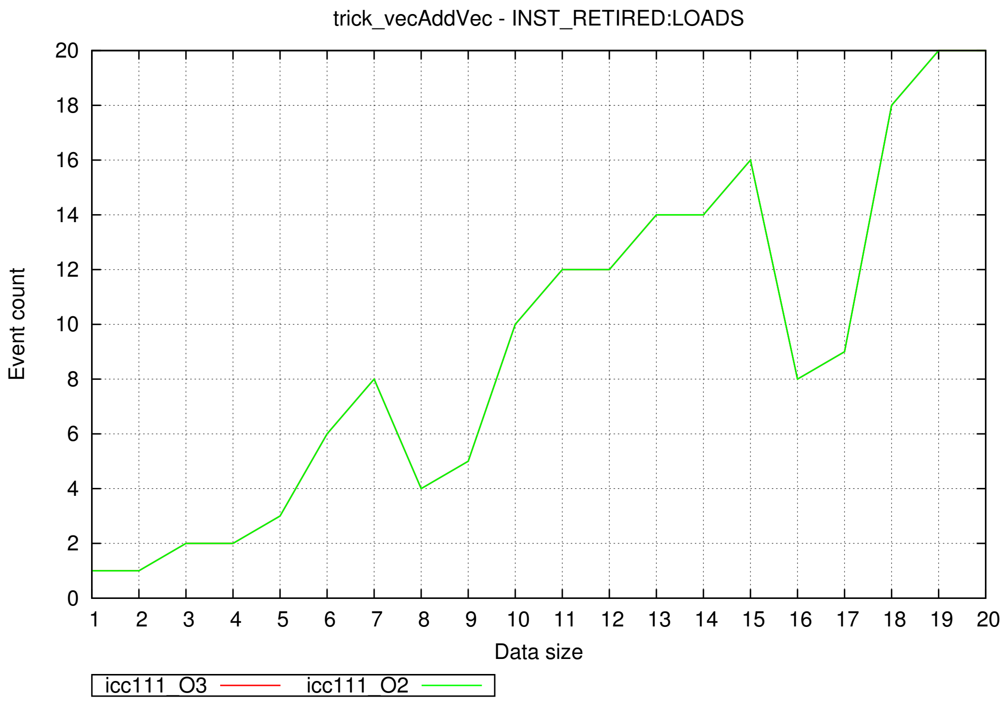

Figure 3.8: Number of loads performed with the fourth volatile design.

3.4 Usage

Only a basic overview is presented of its usage as the framework is in very early alpha state. There are three parts: one for evaluating single programs, one for gathering performance data from many programs and compilers, and one for visualizing data.

3.4.1 Single Program Evaluation

runSingleAndCalulate.sh

This script takes a single program, and makes statistics for a fixed number of loop iterations. The output is a list of non zero event counts that needs to be scaled by the loop multiplication factor inside the program. All example programs currently use a scale factor of 100, so that obtaining the values for a single loop iteration requires dividing by 100.

Sample output:

```
BRANCH_INSTRUCTIONS_RETIRED: 18500
BR_CALL_EXEC: 100
DIV: 500
FP_COMP_OPS_EXE: 11000
INSTRUCTIONS_RETIRED: 159900
INST_RETIRED:LOADS: 26000
INST_RETIRED:OTHER: 124800
INST_RETIRED:STORES: 9600
LOAD_BLOCK:STA: 1226
LOAD_BLOCK:STD: 2952
MUL: 7608
RAT_STALLS:ANY: 2031
RESOURCE_STALLS:ANY: 21705
RESOURCE_STALLS:BR_MISS_CLEAR: 220
RESOURCE_STALLS:LD_ST: 217
RESOURCE_STALLS:ROB_FULL: -48
RS_UOPS_DISPATCHED: 179816
RS_UOPS_DISPATCHED_CYCLES:ANY: 61817
RS_UOPS_DISPATCHED_NONE: 2082
```

```
SB_DRAIN_CYCLES: 14
SIMD_COMP_INST_RETIRED:SCALAR_DOUBLE: 10900
SIMD_INST_RETIRED:ANY: 19300
SIMD_INST_RETIRED:SCALAR_DOUBLE: 19300
SIMD_UOPS_EXEC: 6701
SIMD_UOP_TYPE_EXEC:LOGICAL: 4700
UNHALTED_CORE_CYCLES: 65439
UNHALTED_REFERENCE_CYCLES: 65415
UOPS_RETIRED:ANY: 161924
UOPS_RETIRED:FUSED: 17600
UOPS_RETIRED:NON_FUSED: 144368
```

Note that many values are precisely dividable by 100 while others are not explanations for this behavior is presented in Section 3.3. Also note that RESOURCE_STALLS:ROB_FULL shows a (relative) small negative value, this indicates two issues. The event type is very unstable between runs and the event count in each loop iteration is below what can be measured for that event, without adding more sampling runs. When these events show relatively small numbers, either positive and negative, the real value is most likely zero.

Output can also be written to files, giving 3 files at the same location as the program. The important one, with the statistics, is called *programName.txt*. The two other files contains the raw output from each execution of pfmon.

3.4.2 Parametric Performance Gathering

compileAndCalulateSetList.sh

This script compiles all benchmark programs and benchmarks them for the entire set of compilers and data sizes selected. It was used to generate all the data underlying the graphs in this report. The first parameter is a text file with compiler setup names, paths and names of the compilers and lists of compiler flags to use. The second and third parameters control the data size to iterate over. This script uses a set of other scripts in order to perform its task, delegating each part out to more specialized scripts. This reduces the complexity in each script, and enables faster testing and development. Reruns will not recalculate or recompile existing data, but any completely missing or new benchmarks will be performed.

verifySetList.sh

This script checks for the existence output files generated by compileAndCalulateSetList.sh, and reports any missing file after a faulty, incomplete or canceled run. It can also optionally remove incomplete files, enabling compileAndCalulateSetList.sh to fill in the missing parts.

recalculateSetList.sh

If different statistical tools are to be used (or a bug is found in the current design), this script will take all the raw data from every benchmark run and recalculate the statistics.

3.4.3 Visualization

makePlotSet2.sh

The script was used for creating all the graphs in this report. All the performance data must first be generated with compileAndCalulateSetList.sh in order to use this tool. The first time a compiler/program combination is used a cache is created, enabling quicker graph generation later on.

The first parameter is the name(s) of compilers setups that are to be used for plotting the graphs. The second parameter is a simple regexp to select one or more benchmark program names. The third (optional) parameter selects which event type to show, defaulting to UNHALTED_CORE_CYCLES if not given. This parameter can either be a preselected name (like cpi, mathop, mathopf, packedratio, ...), the name of an event, or the event index number. Aggregated event types can also be constructed by combining several events into more complex expressions.

Examples with an interactive window output:

```
$ ./makePlotSet2.sh "icc111_O2 " invert

$ ./makePlotSet2.sh "icc111_O2 " invert cpi

$ ./makePlotSet2.sh "gcc..._O2 " invertP INSTRUCTIONS_RETIRED

$ ./makePlotSet2.sh "icc110_O3" invertP (\$1)

$ ./makePlotSet2.sh "icc111_O. " invertP "(RS_UOPS_DISPATCHED
/(UOPS_RETIRED:ANY+UOPS_RETIRED:FUSED)-1)"
```

Example with postscript output is turned on, giving pdf files for reports (must be enabled inside makePlotSet3.sh):

```
$ ./makePlotSet2.sh "icc111_O. " invertP cpi && epstopdf out.ps
 --outfile=invertP_cpi.pdf
```

3.5 Examples

Two examples are used to demonstrate various possibilities, both using ICC 11.1. The first example requires that auto-vectorisation is used with the smallest data sizes, and unfortunately this is not the case for GCC - as it seems to have a hardcoded limit preventing auto-vectorisation with the code used here (multiple attempts were made to change this behaviour, but the relevant flags either gave no effect or generated errors). The second example uses ICC 11.1 only by chance, but it should not affect the analysis.

3.5.1 Vector-Vector Addition

Evaluation of ICC 11.1 using -O3 on an old version of the vector-vector addition test program (using the third volatile design). First look at the performance graph in Figure 3.9.

Figure 3.10 and 3.11 shows that only the expected amount of data is loaded and written, increasing stepwise as scalar operations are replaced by packed. Note that that all the data from both vectors are loaded, not only one.

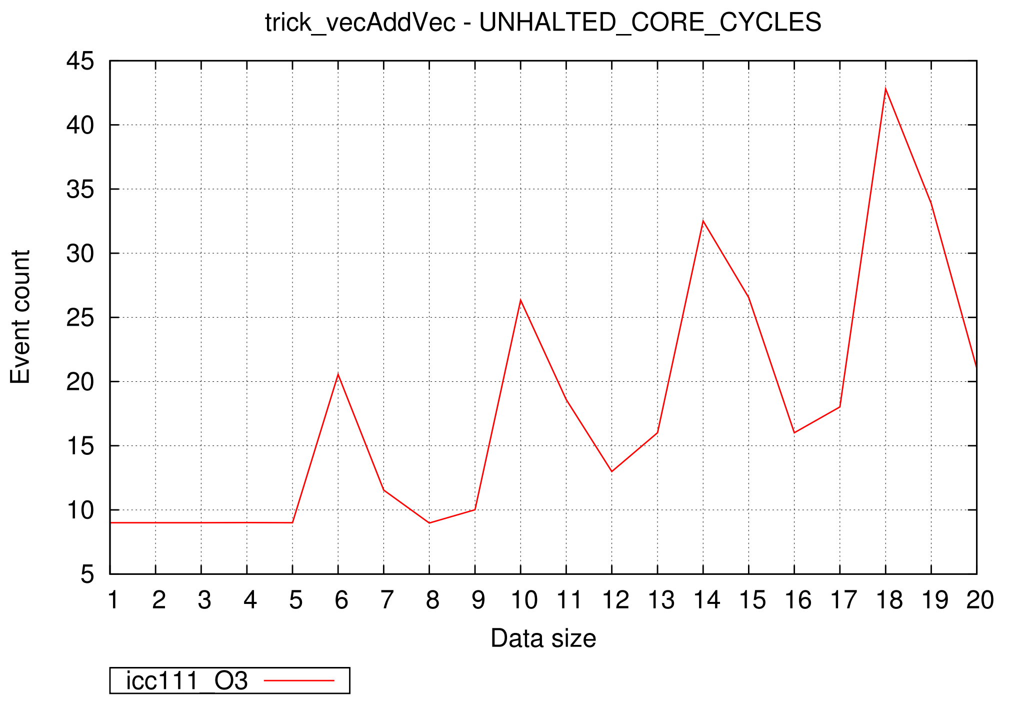

Figure 3.9: Very strange performance. The spikes and the very flat region at the start.

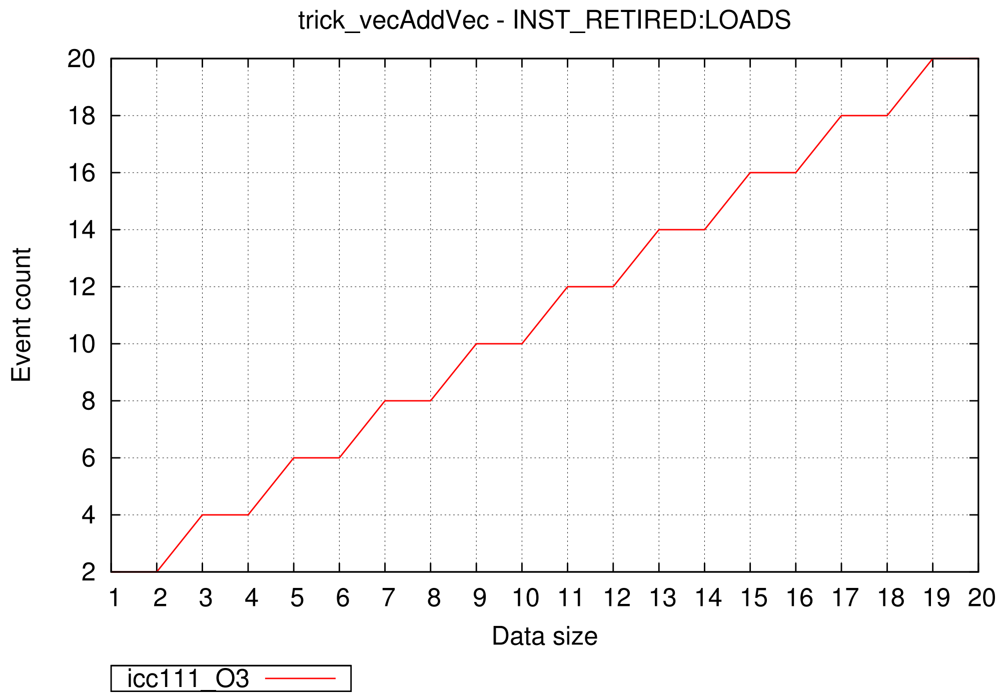

Figure 3.10: All the loads are there, one for each vector, but no excess.

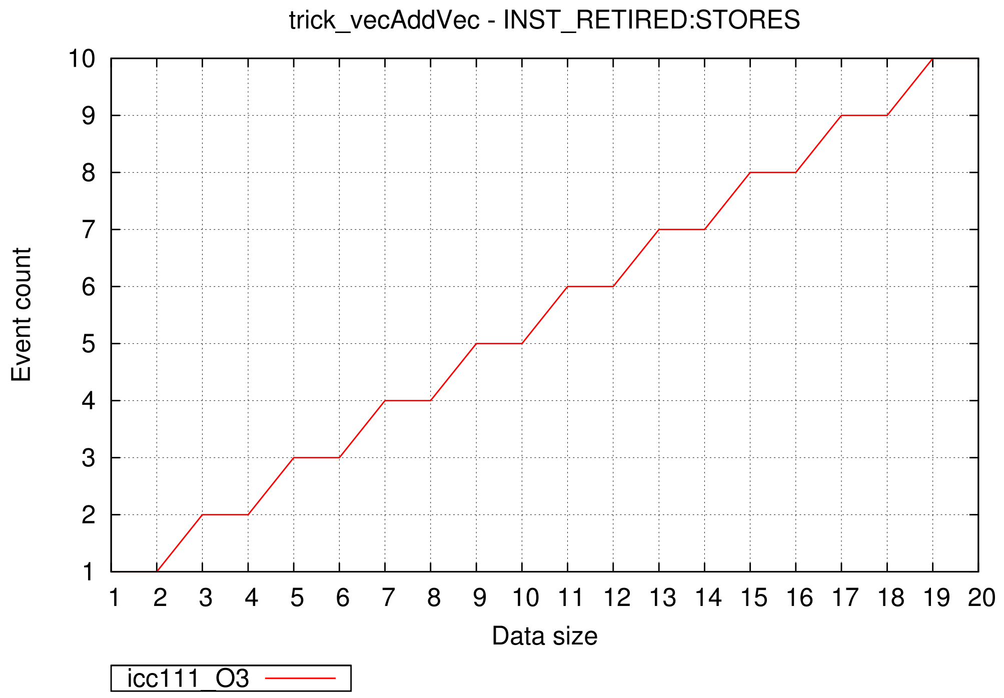

Figure 3.11: All the stores are there, one for the answer vector, but no excess.

All the math see is performed too, as shown in Figure 3.12. But a clue can be found by looking at the number of other instructions (not loads/stores) performed; Figure 3.13. Note that for some sizes its very low. Math instructions are fused with loads; see Figure 3.14. It seems all the math instructions are fused, so there are some other instructions being performed.

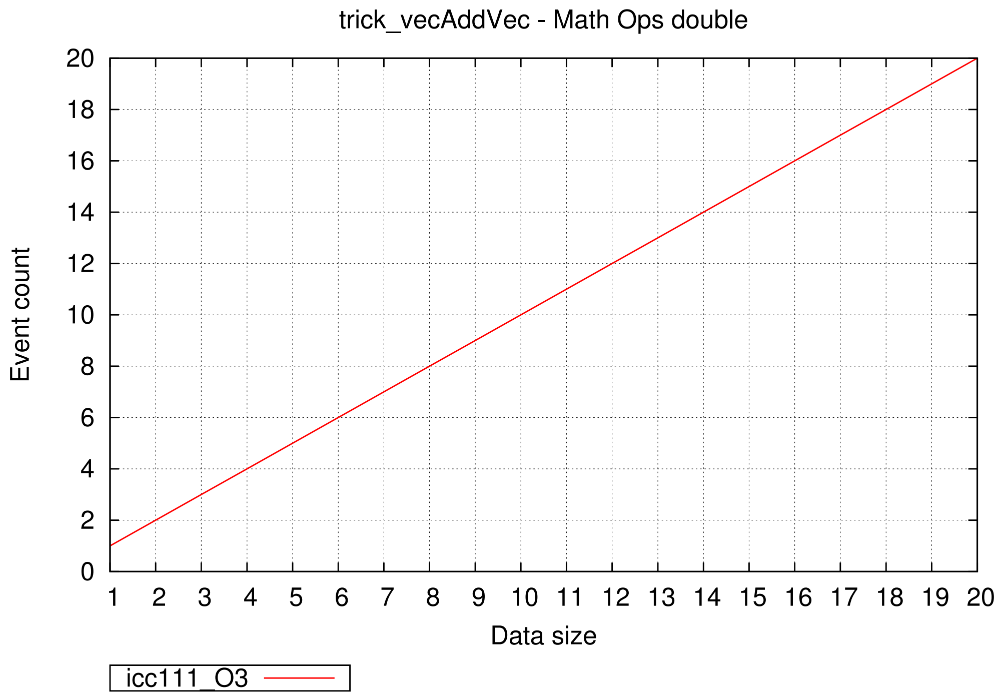

Figure 3.12: Verification of the count of math operations performed.

Figure 3.15 shows that many of the excess instructions are branch instructions. This means that branches are the reason for the low performance, a single vector-vector addition is not always unrolled. But this should only create a performance graph similar to the branch instruction count, and not the sharp spikes (as seen in Figure 3.9). Looking at branch miss-predictions, in Figure 3.16, we see that they match the sharp spikes. Those misses give a performance cost that comes in addition to the extra instructions used by the branches. Now the spikes are explained, but not the flat performance for small vector sizes.

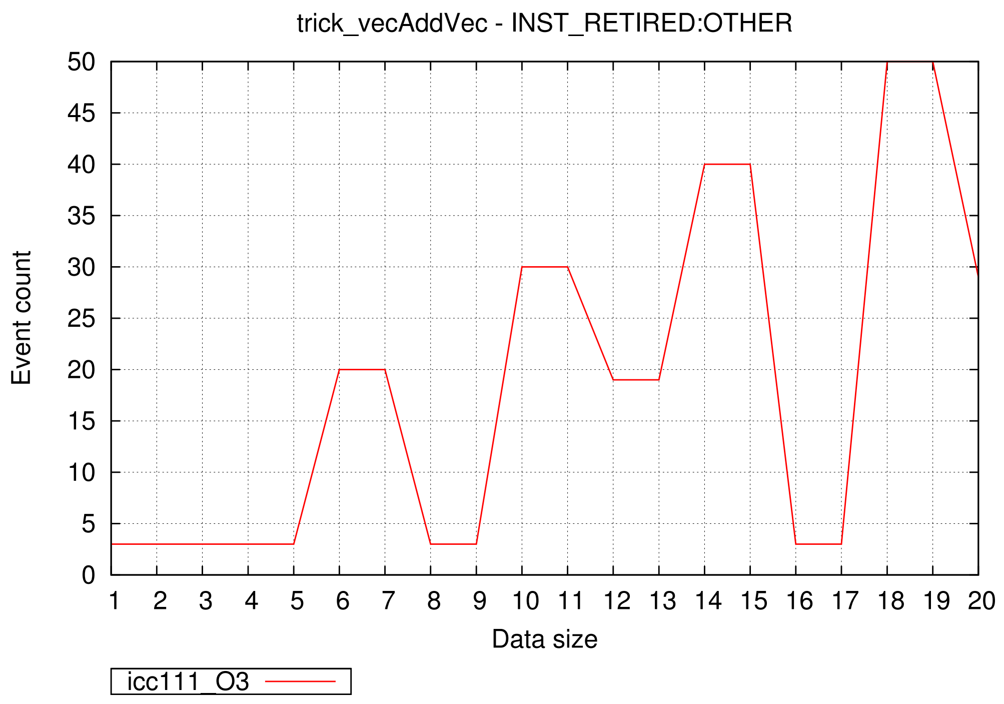

Figure 3.13: Count of other (non load/store) instructions.

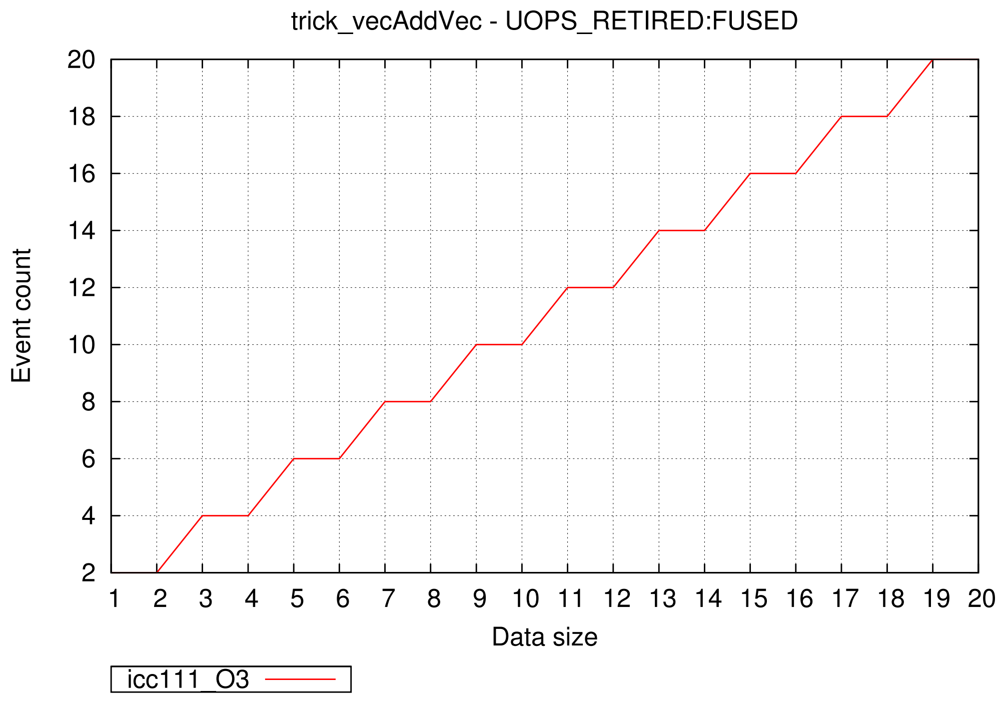

Figure 3.14: Count of fused instructions, here math and load are fused.

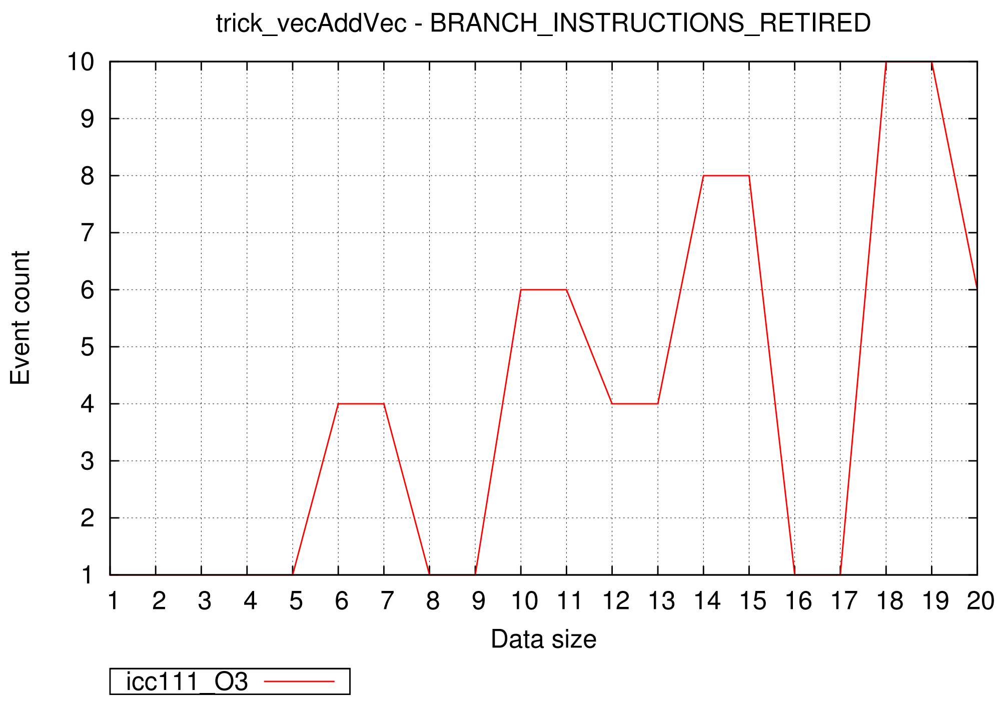

Figure 3.15: Count of branch instructions, indicating non-unrolled code.

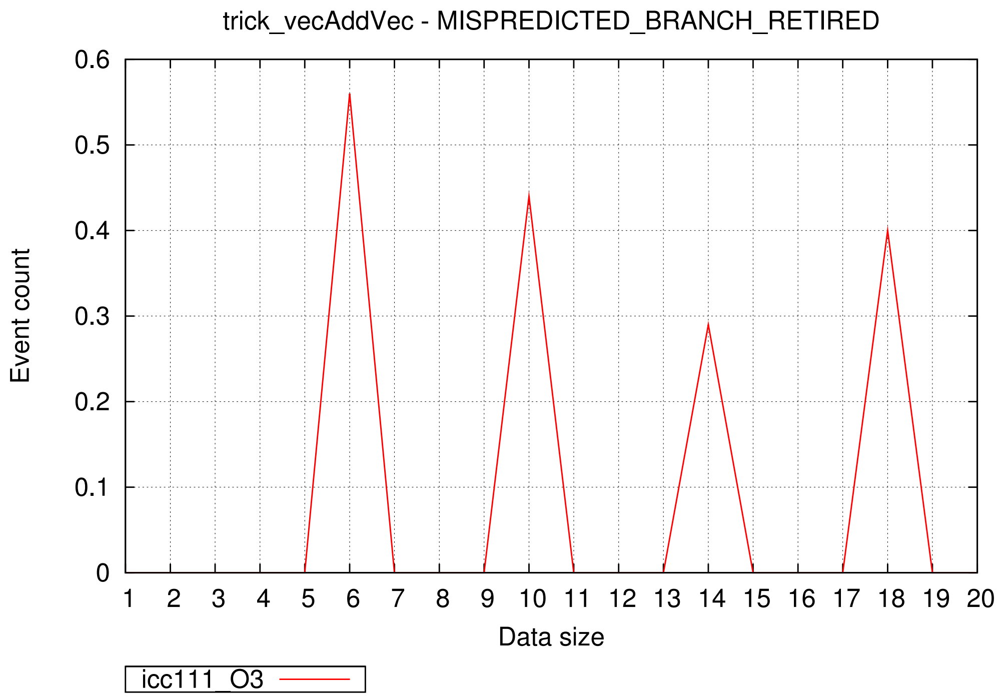

Figure 3.16: Count of branch misses.

In Figure 3.17 the number of blocked loads is shown. This is a key factor for explaining the fixed performance for the small data sizes. When one stores and loads rapidly from the same location, there can be performance issues. With the small vector sizes this becomes a limiting factor. The loads stall because they need data written in the previous iteration. The extent of the stalls can be seen in Figure 3.18, showing the number of cycles without any useful work (started). Note that these are absolute values and not scaled in any way.

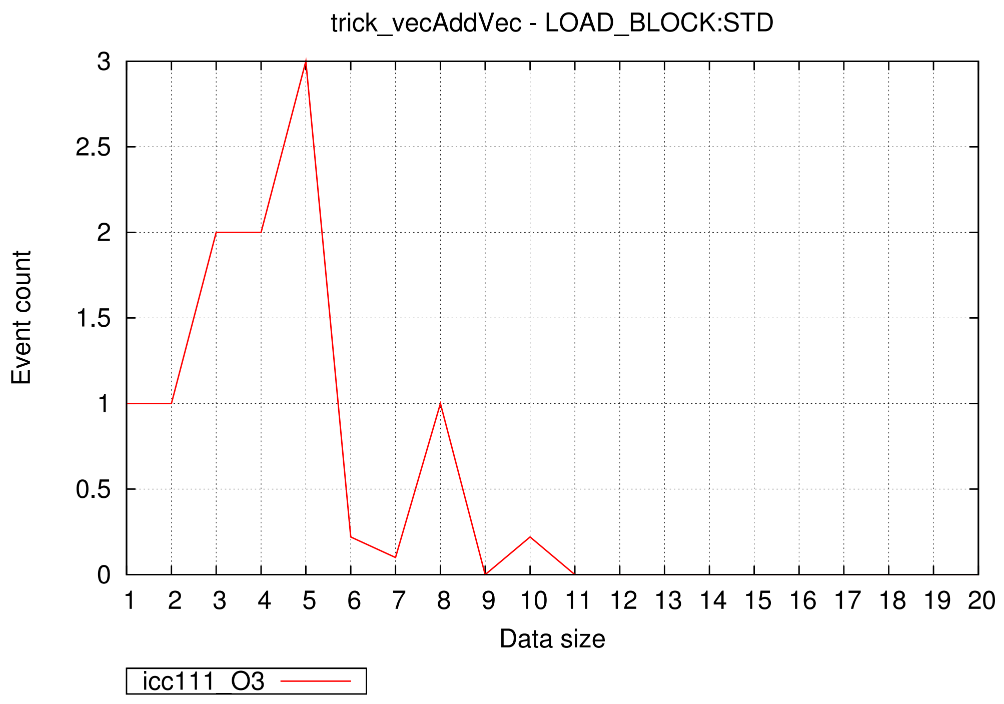

Figure 3.17: Count of load stalls rereading data recently written.

Looking into the details of the Out of Order Engine is also possible. Figure 3.19 show how often any type of resource stall occurs. In Figure 3.20 the number of times there are too few resources for handling memory load instructions is shown. When there are no extra instructions related to branches it seems that there might be too many load instructions (size 8-9 and 16-17) .

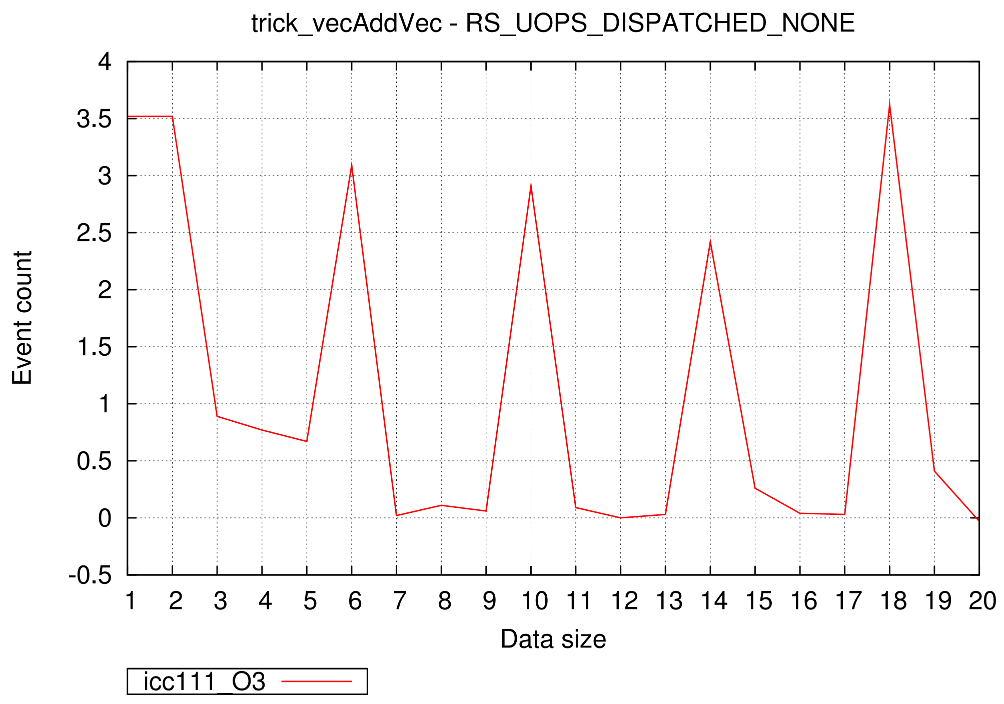

Figure 3.18: Count of cycles without any dispatched instructions.

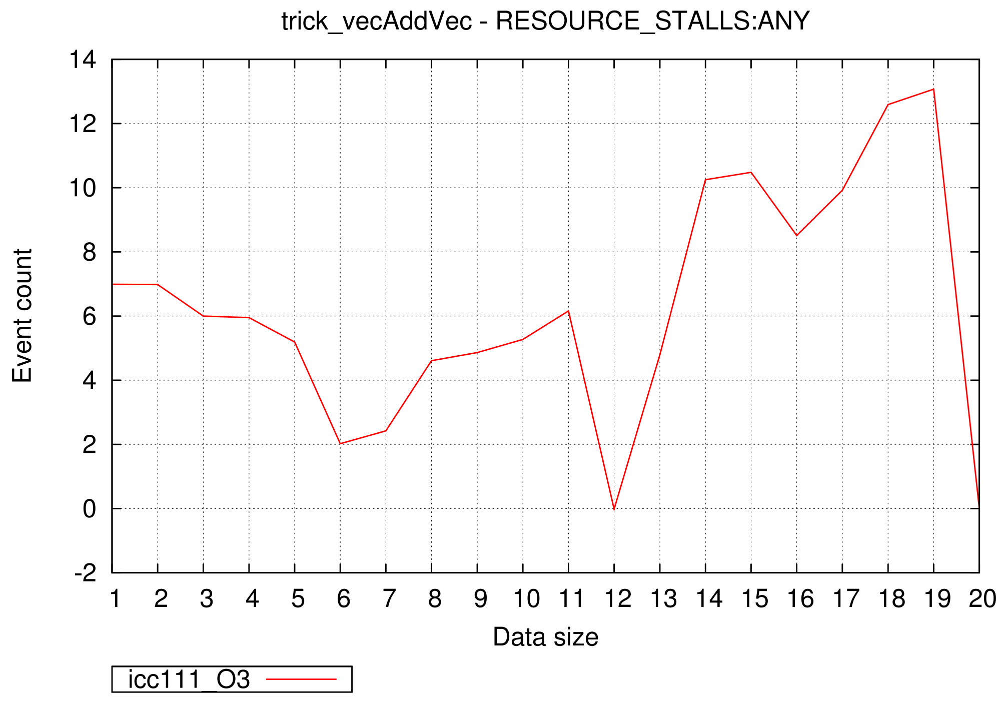

Figure 3.19: Number of *any* resource stalls.

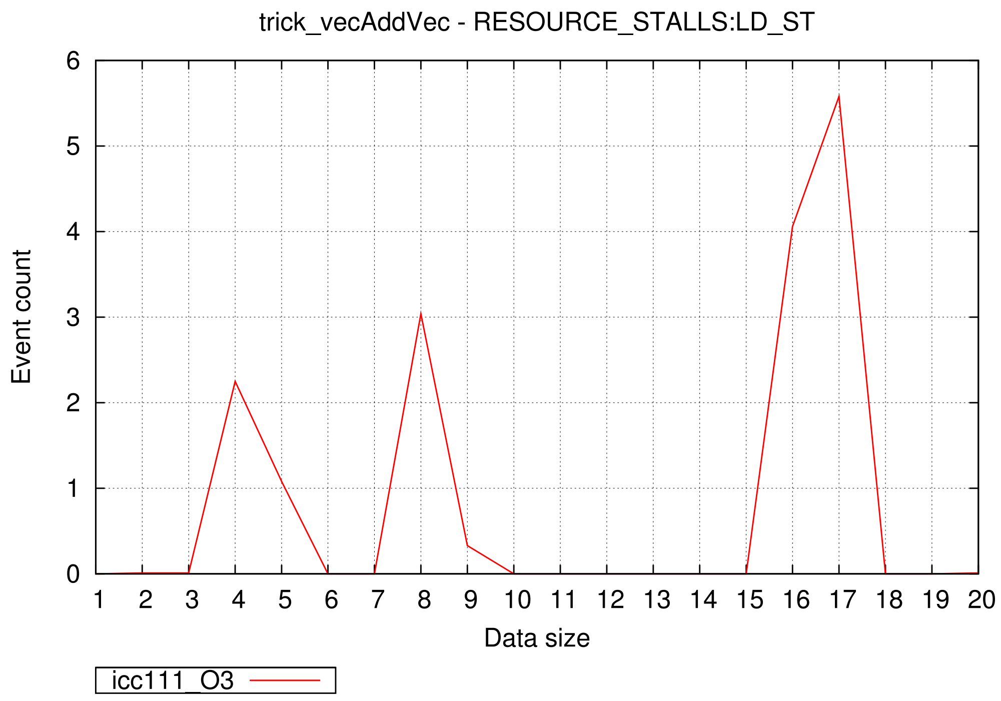

Figure 3.20: Count of load port resource stalls.

3.5.2 Inversion

Taking a different example, inversion of matrices, several possibilities are shown. There are three versions of inversion in SMatrix, and it is useful to compare them using a symmetrical matrix. The first algorithm 'Fast' gives somewhat lower precision using specialized code for small matrix sizes (the default Bunch-Kaufman (B-K) is used for larger sizes). The second is based on Cholesky decomposition, using specialized code for small matrices and more general code for larger sizes. The last is the default B-K which has no specialized code paths. Figure 3.21 show that the CPI is neither very bad or good, and the number of cycles in Figure 3.22 show no issues either.

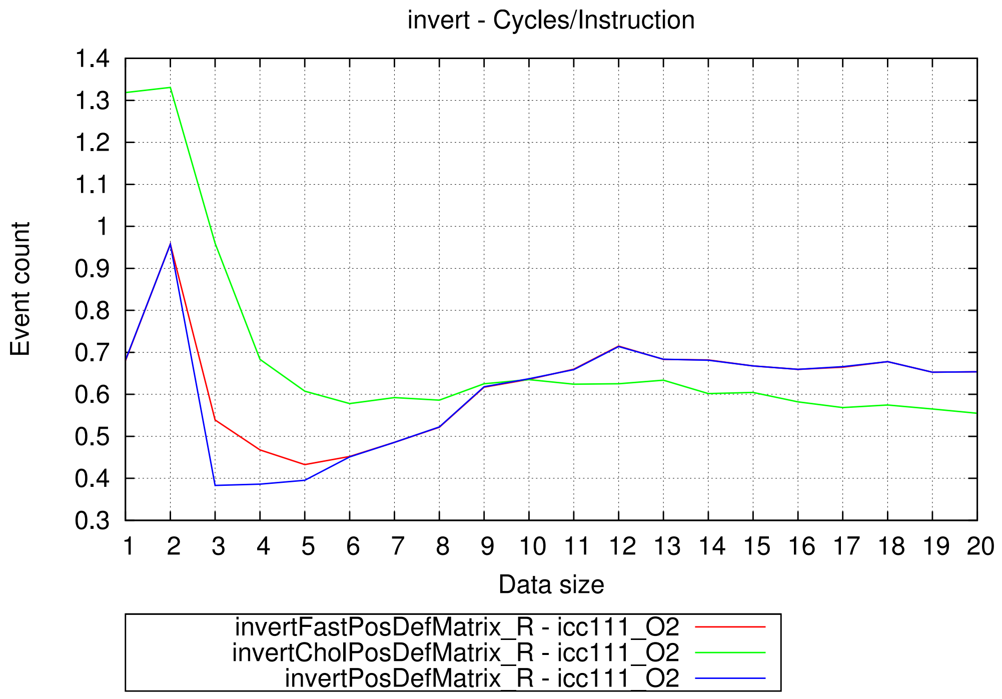

Figure 3.21: The CPI for three versions of symmetrical matrix inversion.

This basic analysis show that there is no obvious problem with the code, but it is not very efficient. The smallest data sizes are not very efficient, as expected. More in-depth evaluation can then be performed. Figure 3.23 show that the CPU is wasting time, performing no work.

Figure 3.24 shows that a large fraction of the uops started are later cancelled, representing useless work.

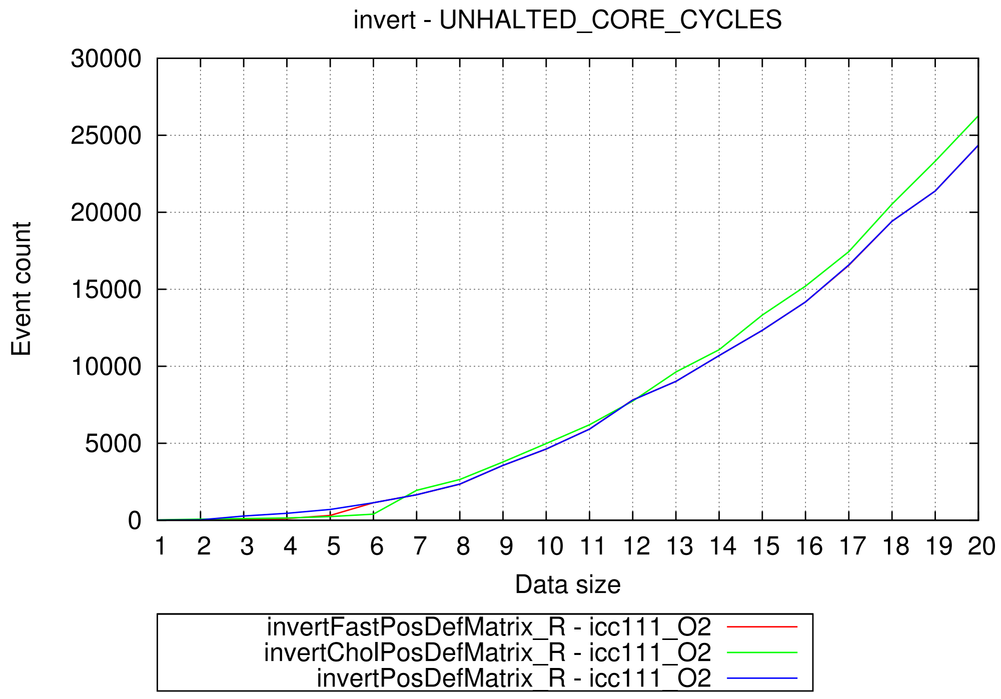

Figure 3.22: Processor cycles spent to perform a single invert function.

The first candidate to check for generating all the useless uops is misspredicted branches. Figure 3.25 shows there are many.

Looking at the ratio of mispredicts per instruction in Figure 3.26 shows a very good correlation with the wasted work.

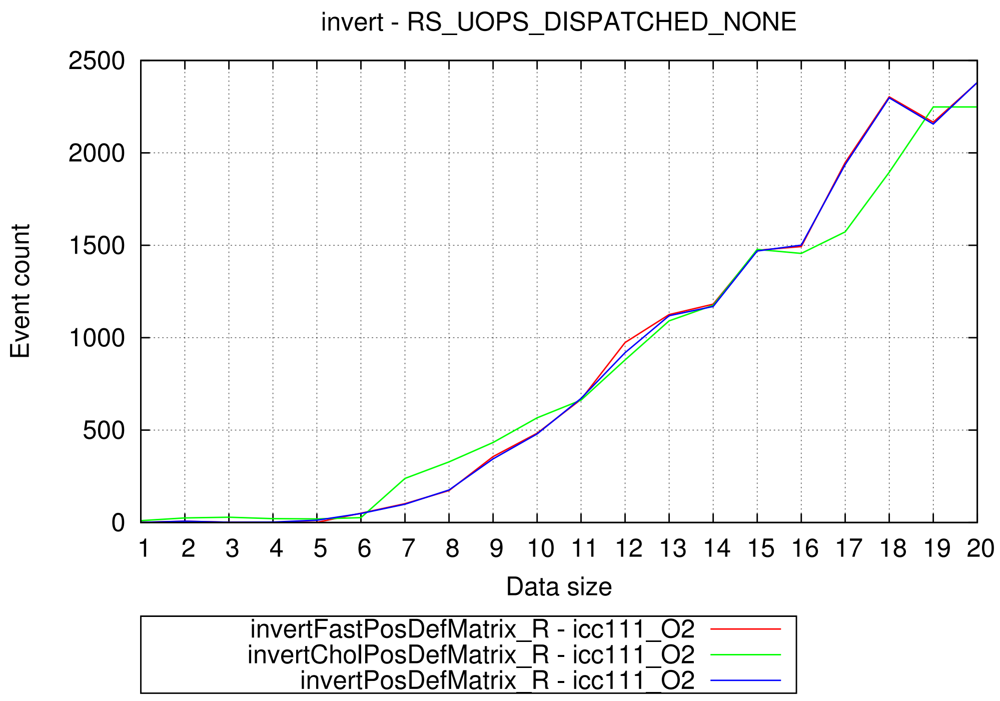

Figure 3.23: Cycles without any dispatched uops.

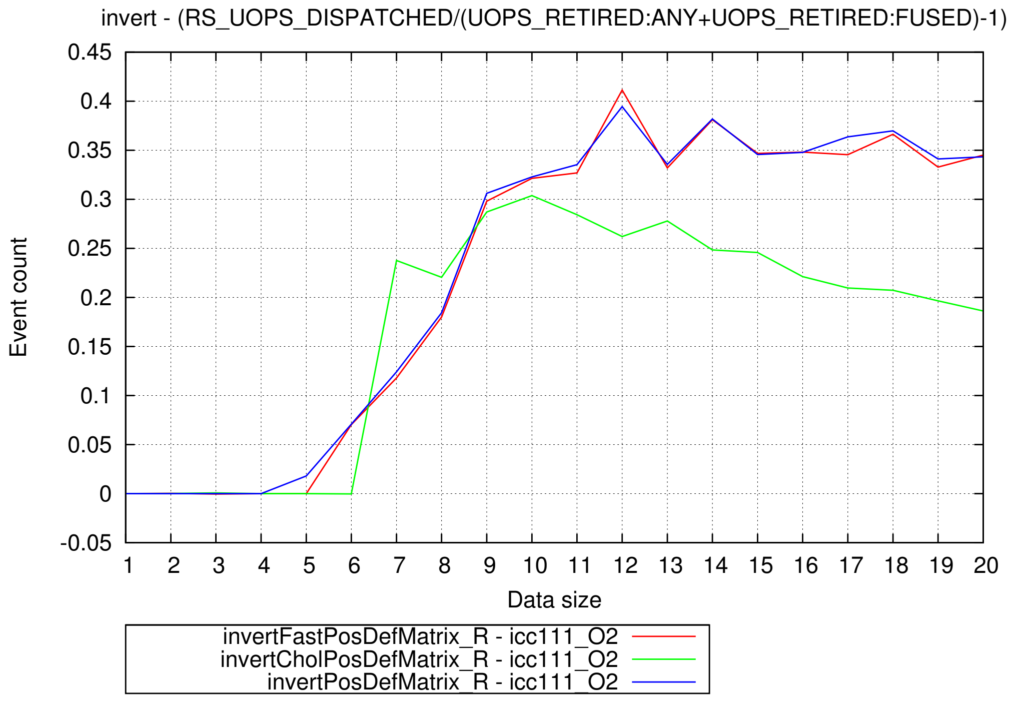

Figure 3.24: Fraction of uops that are started, but cancelled before completion.

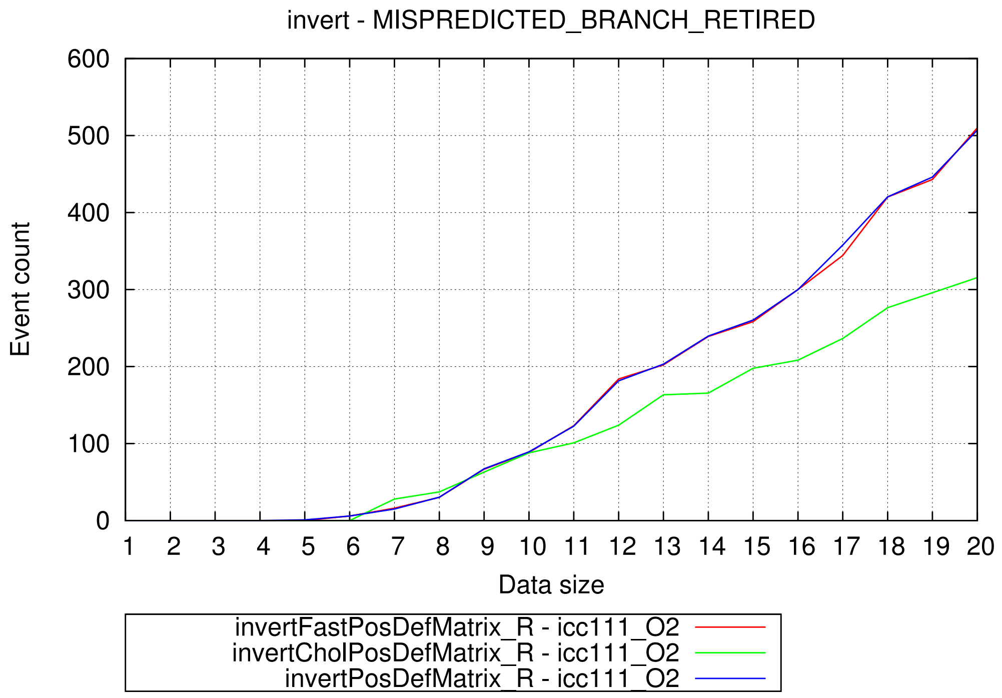

Figure 3.25: Counts of branch misses.

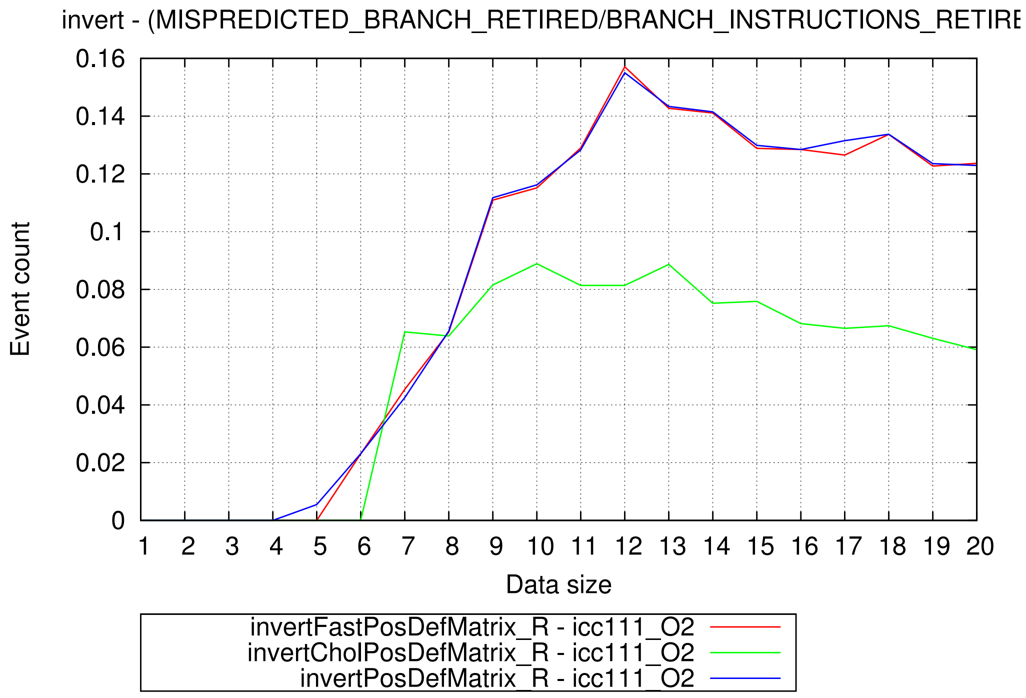

Figure 3.26: Branch misses/branch ratio.

The CPI profile (Figure 3.21) does match the amount of useless work and mispredictions if one excludes data size 1-2 where overhead and the tight loop skews the results. This indicates that they most likely prevent the CPI from improving. Figure 3.27 shows the fraction of cycles where no new uops were started. This graph matches the CPI, and gives a good indication of the main performance limitation.

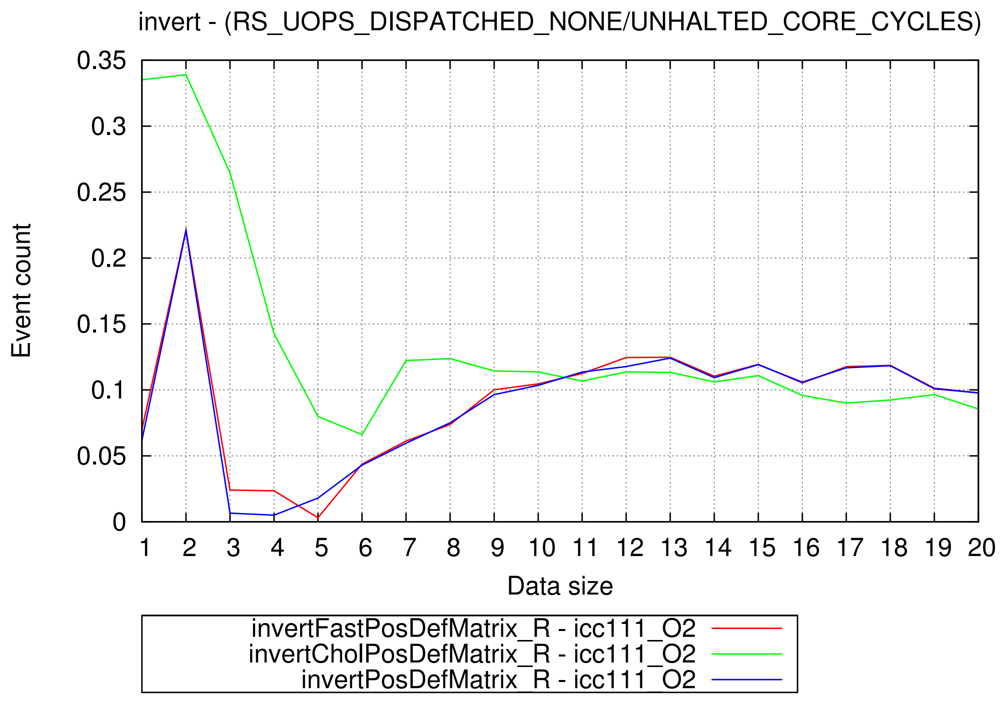

Figure 3.27: Fraction of cycles where no new instructions/upos are issued.

More analysis is possible. Looking at the ratio of SIMD (SSE) instructions in Figure 3.28 indicates that there is much overhead as well as all float/double computations and moves should use SSE instructions. Both the ratio of load instructions (Figure 3.29) and store instructions (Figure 3.30) are low. This indicates that some different type of instruction is the main part, and this can be seen in Figure 3.31. Identifying the issue involves more decomposing of the instructions. Figure 3.32 shows the ratio of branch instructions. This is a relative high amount considering that branches often requires 2-3 instructions, one for the branch itself, one for a test/compare and one for an increment counter. While the test/compare can be avoided, there are normally instructions related to pointer arithmetic and register setup in or before loops. This indicates that removing the mispredictions might improve the CPI, but it might also be better to reduce the number of loops.

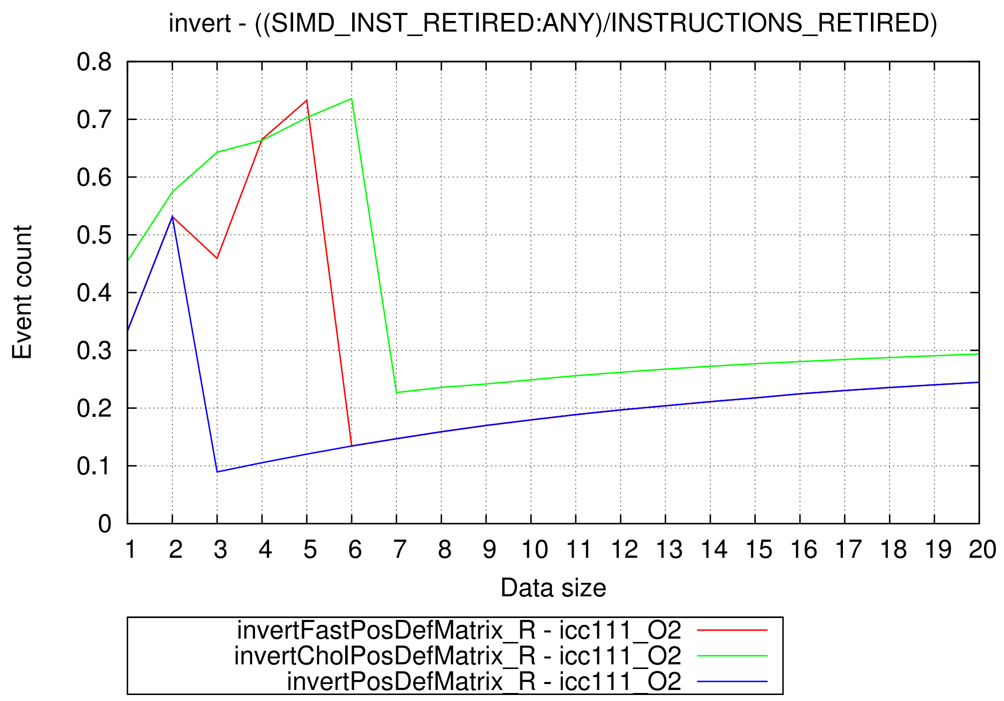

Figure 3.28: SSE instructions pr. any instruction.

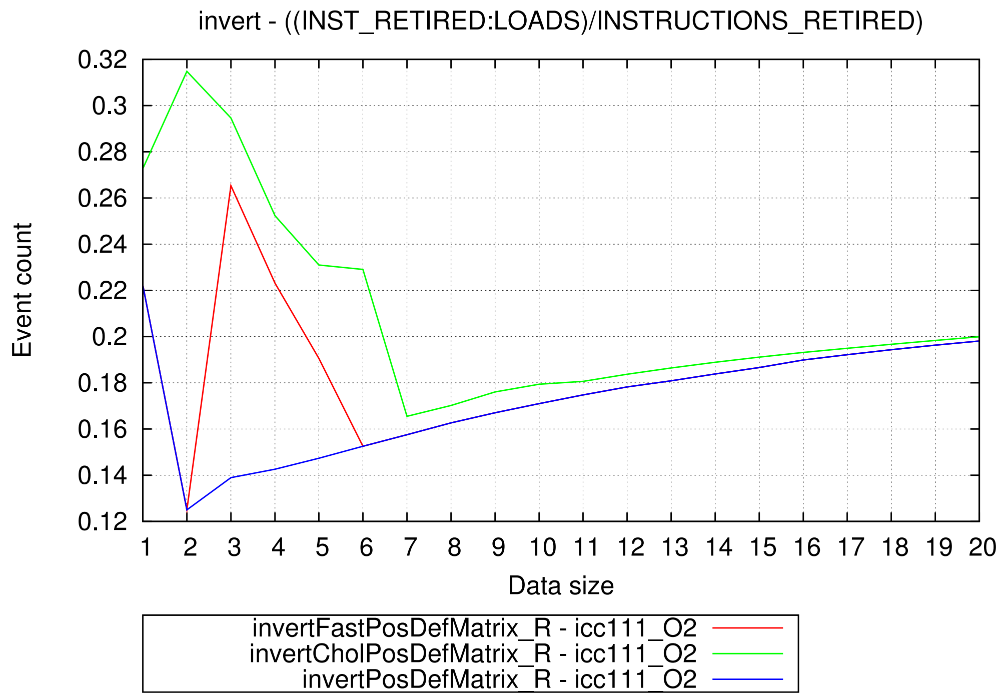

Figure 3.29: Load instructions pr. any instruction.

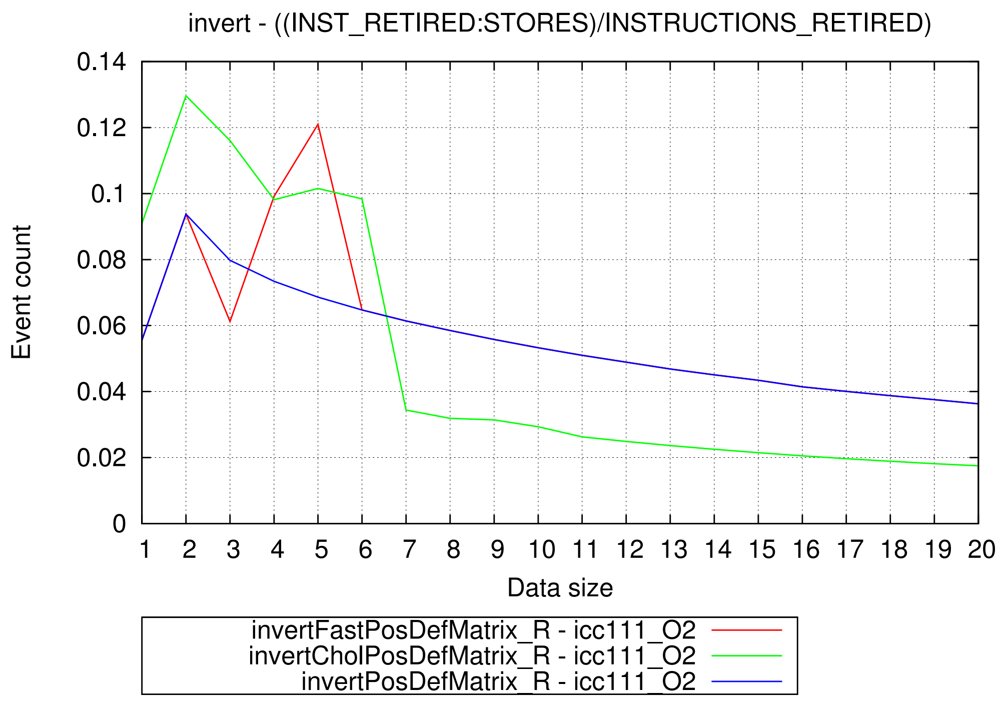

Figure 3.30: Store instructions pr. any instruction.

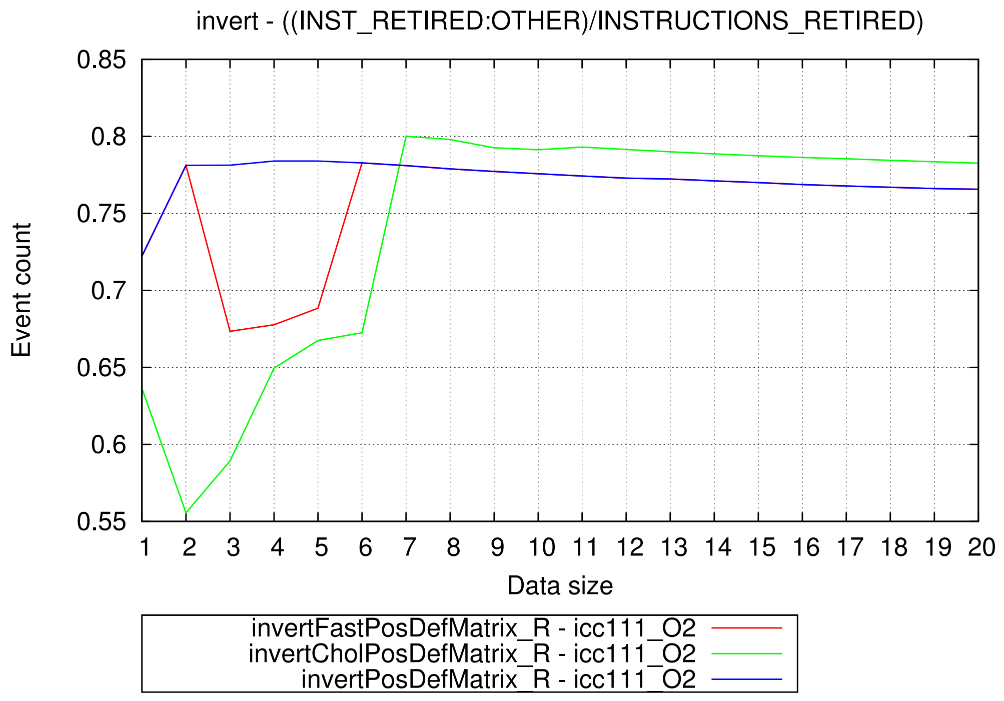

Figure 3.31: Other instructions pr. any instruction.

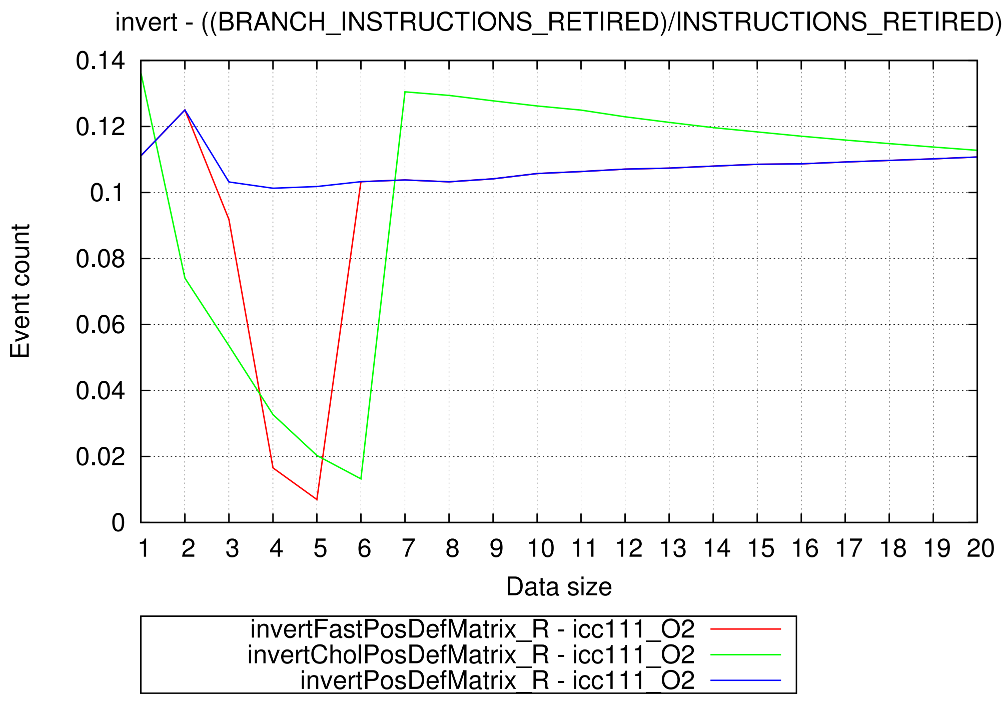

Figure 3.32: Branch instructions pr. any instruction.

Reducing the loop count might help and can be tested using various compiler flags. Figure 3.33 and 3.34 show invertPosDefMatrix_R, and going from O2 to O3 reduces both the number of branches and mispredicts. Figure 3.35 shows the performance, and compared with either Figure 3.33 or Figure 3.34 it is clear that mispredictions explain the performance difference best. Note that this analysis for comparing O2 and O3 is not complete, and is included only to demonstrate possibilities.

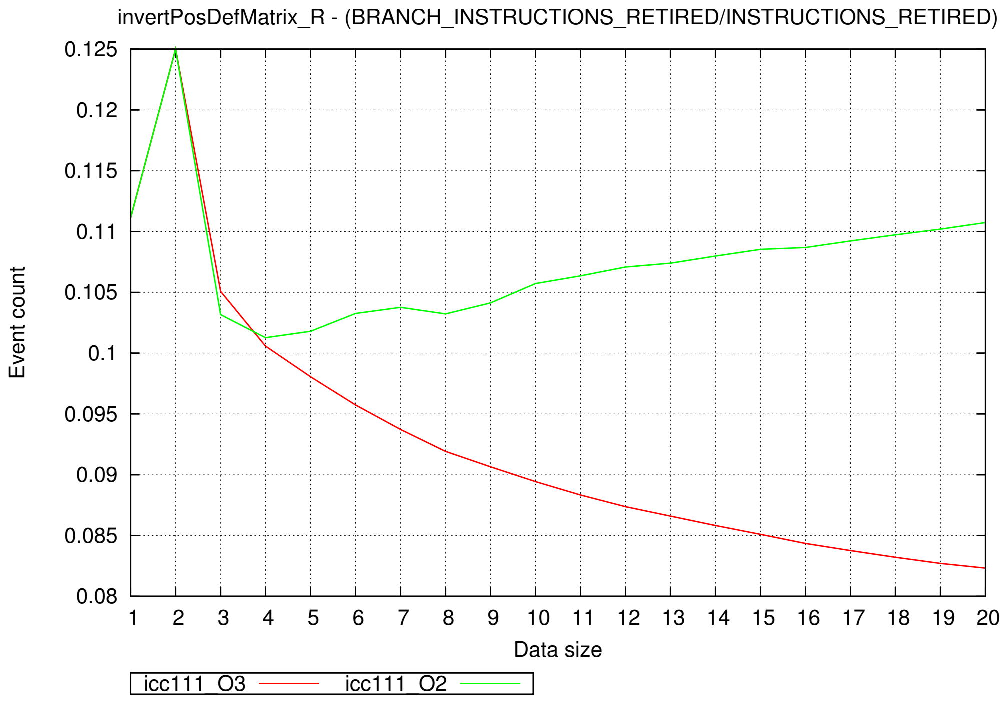

Figure 3.33: Branch instructions pr. any instruction, with O2 and O3.

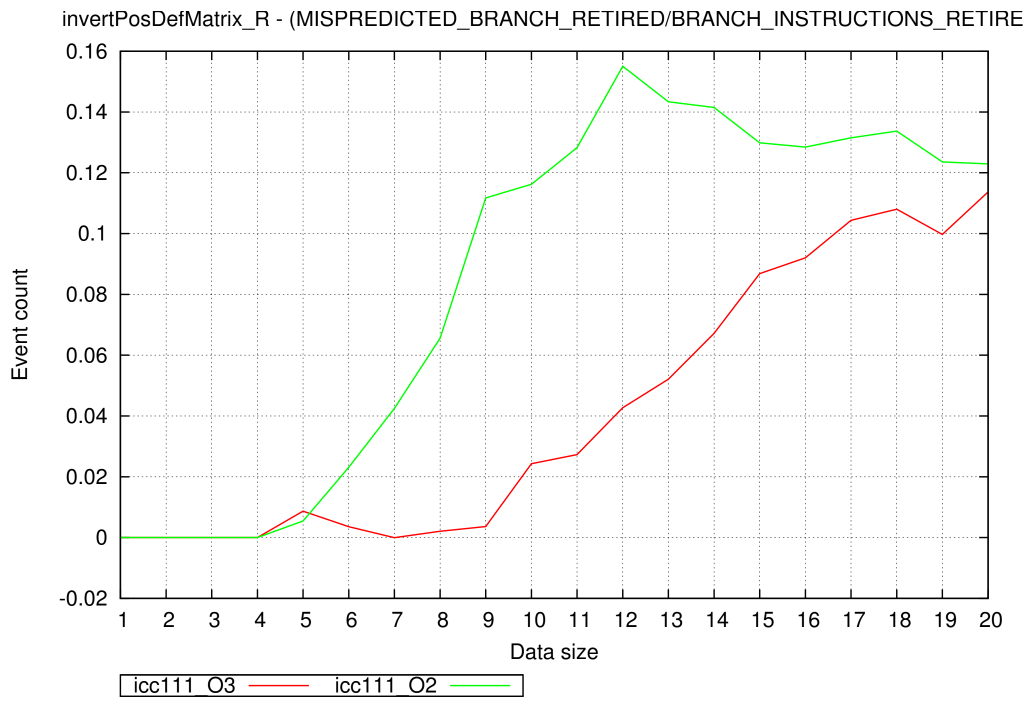

Figure 3.34: Branch miss pr. branch, with O2 and O3.

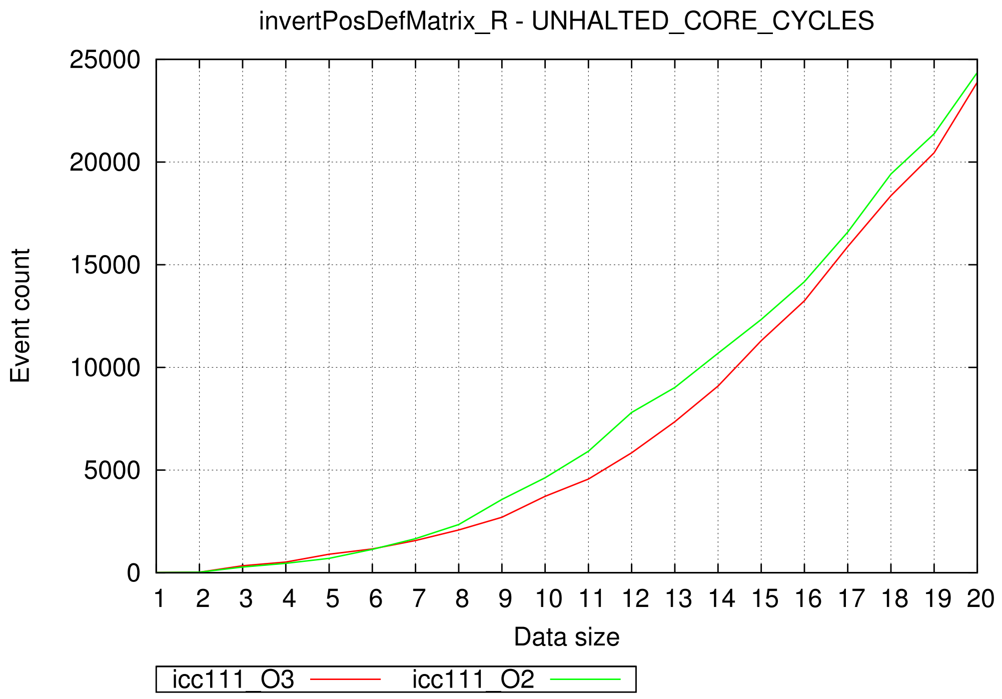

Figure 3.35: Branch miss pr. branch, with O2 and O3.

3.6 Limitations

There are several limitations in the design and method used. The single biggest limitation is that the benchmark programs must be designed to fit the framework, allowing control over loop counts. The second limitation is that numerous runs are required in order to obtain all event types with good accuracy. This limits the usability to smaller programs and tests.

A number of unsolved issues also exist, relating to the early stage of development. There are several parts that use hard-coded values (like paths), even while some are read from configuration file(s). There is currently no code size detection, meaning that optimization and insight to instruction size are missing. The feedback from compilers during compilation needs to be analyzed, enabling graphs of auto-vectorization successes/failures. A simple database needs to be utilized in order to handle the relatively large amounts of data generated, currently more than 900MB of text files.

Too few events might be sampled, useful events might not sampled at all, or be sampled to few times too eliminate fluctuations. A larger event set is needed both for detailed analysis of problematic programs, and to gain more insight into the nature of various events. Specifically, no event related to the cache, data bus, Translation lookaside buffer (*TLB*) misses and multiprocessor effects is currently used.

3.6.1 Performance Counter Bugs

A different type of problem is the accuracy and correctness of the performance counters. During development several bugs of varying severity were encountered. Some events have only inaccurate descriptions, while others even count the wrong events. These issues have significantly slowed development as each bug had to be found and verified in the disassembled binaries, test programs written, and bug reports created. Because of the nature of performance counter events it is advisable to carefully evaluate data (and the description) from each new (unfamiliar) event, expecially on new processors. If an event seems to behave in an unexpected way then detailed analysis, with specially designed test programs, is recommended.

3.7 Current and Future Work

With the current raw but functioning implementation, one of the most important requirements is a rewrite, both to remove significant script overhead and increase usage flexibility. The possibility to dynamically control both the number of iterations, and select which events to monitor is needed. Having this kind of dynamic evaluation might significantly improve the data quality of unstable events by adding more runs, and reducing the required runtime by eliminating needless (re)counting of events.

Having all the data, the raw output from pfmon (from each run), the processed values (for one program), version information of the include files (name and MD5 sum), the (detailed) output from the compiler, compiler flags, the C++ source code and the assembly language code (.s files) easily available and searchable will enable a multitude of rapid analytical insight. Automation of regression analysis, compiler efficiency trends, bug detection and compiler flag suggestions is possible. With newer compiler support for function specific optimization, automatic flag search and associated *#pragma* control generation might be performed as well.

The addition of runs with statistical sampling indicating where in the code issues appear is also possible. Merging the noisy statistical sampling data with the accurate counts might significantly improve the value of both data types. By using the precise count values, the scale and distribution from statistical sampling could be corrected, giving a much more accurate picture. Also the sampling intervals could automatically be selected to match the number of events in the code, enabling the sampling triggers to occur either at selected locations, or with periods providing statistically balanced sampling points.

Chapter 4

ROOT

Much work has been performed on the SMatrix and SVector package in ROOT[1]. Some of the changes and performance evaluation will be described in this chapter. Only a brief overview of the work performed will be presented.

4.1 SMatrix and SVector Standalone Code

A number of standalone benchmark programs were designed; they fall into two categories. The first are the ones that are extracted from the original SMatrix and SVector benchmark programs. They have been made independent from the rest of the ROOT package. This required both extracting the relevant include files, and modifying the source code in the benchmarks. Also, the timer functions used were deeply integrated into ROOT, so new code was written for the timing functions. A total of 4 programs were converted in this way, and another benchmark program (with the same style) was created as well.

A second set of benchmark programs was created for the framework described in Chapter 3. Here a total of 21 different programs were created. Most of these programs are very simple, and have a name indicating the function performed. Some of them are described in more details later.

[1]`http://root.cern.ch/`

4.1.1 Compiler Setups

Each of the standalone programs has been tested with a set of compilers. With a total of 63 compiler configurations, the most common cases have been covered. 29 configurations exist for GCC, spread over versions 4.2.4, 4.3.3 and 4.4.0. 32 configurations exist for ICC, spread over versions 11.0 and 11.1. Only 2 are used for LLVM[2], mainly as it was only added for reference and there was no time to investigate its options. For OpenCC there are unfortunately no configurations in the latest run, as one of the programs (with certain datasizes) generated an internal compiler assert error. Adding support for single missing tests was not possible because of time constraints, so it is currently left out.

4.1.2 Performance Tests

There are 4 sets of basic matrix multiply, namely *C=A*B* and *C=A*Transpose(B)*, in both double and single precision using square non-symmetrical matrices. Intended to evaluate performance of transpose, notably a speedup should be possible. Figure 4.1 and 4.2 show the in cycles needed for the double and single precision respectively. Only sizes 1-5 are shown as this range is the most common. Note that the performance difference between floats and doubles is quite small.

There are several versions of basic vector-vector addition, as shown in Chapter 3. Two additional versions are hand coded using intrinsics, using the same constraints as the fourth volatile design. The first uses aligned loads and stores, which are faster, but requires that the data structures are aligned to 16 bytes. The other design is identical except that loads and stores are unaligned, representing the possibility that alignment guarantees do not exist. Figure 4.3 show the performance difference between the two, the vectors are aligned in both cases.

[2] `http://llvm.org/`

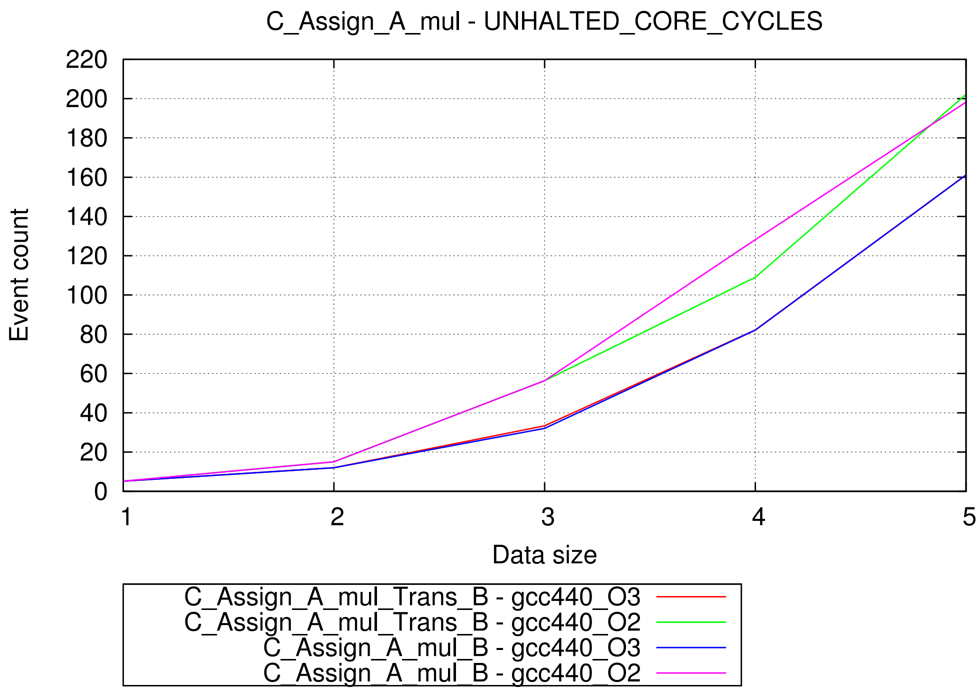

Figure 4.1: Performance of *C=A*B* and *C=A*Transpose(B)*, double precision.

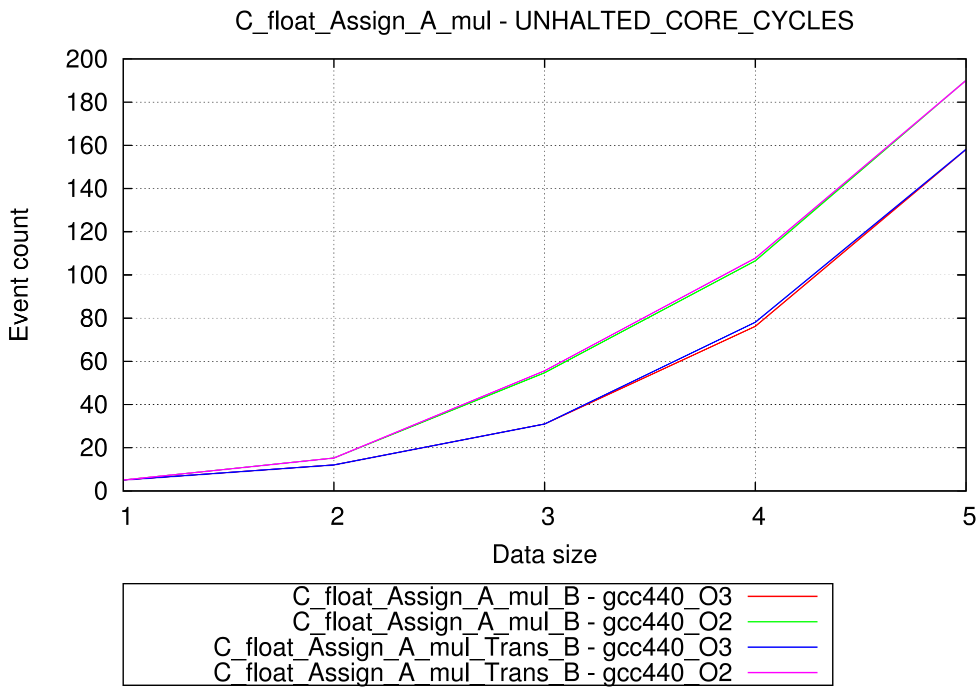

Figure 4.2: Performance of *C=A*B* and *C=A*Transpose(B)*, single precision.

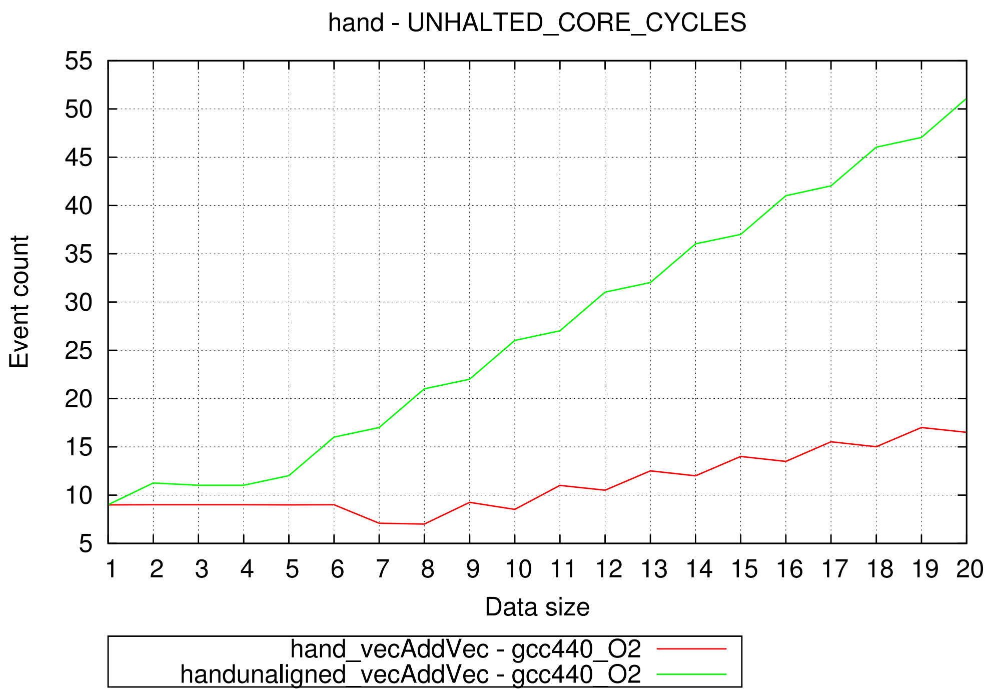

Figure 4.3: Vector-vector addition using intrinsics with identical design, except the alignment requirement.

Several parts of the Kalman filter (CMS version) were decomposed into separate test programs giving insight to which parts can benefit most from optimizations (or a different compiler). Figure 4.4 show the cycle cost for these parts, Figure 4.5 shows the CPI, and Figure 4.6 shows the possibly more important Cycles Per Math Instruction (CPMI). Note that the CPMI might not be a good measure in all of the cases, see Chapter 2.2 for more information.

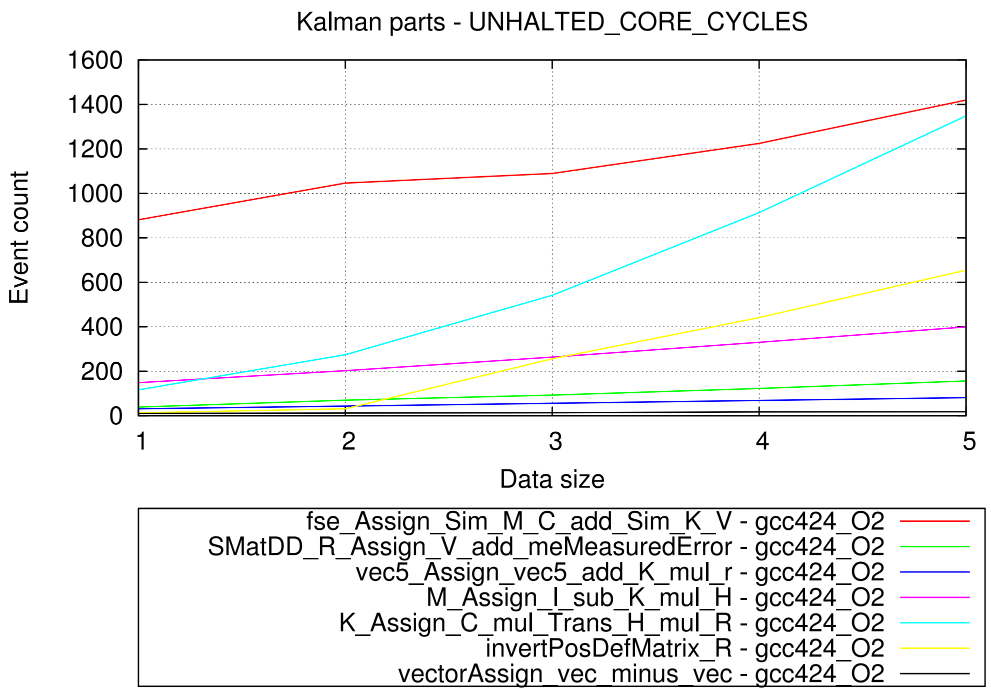

Figure 4.4: Cycles spent on different parts of a Kalman filter.

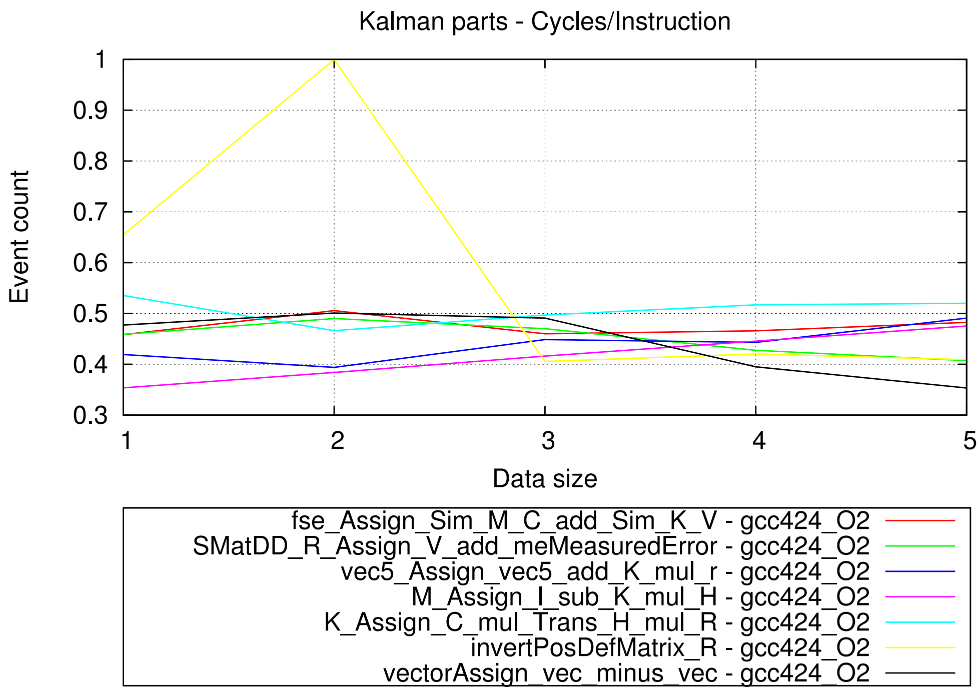

Figure 4.5: The CPI of different parts of a Kalman filter.

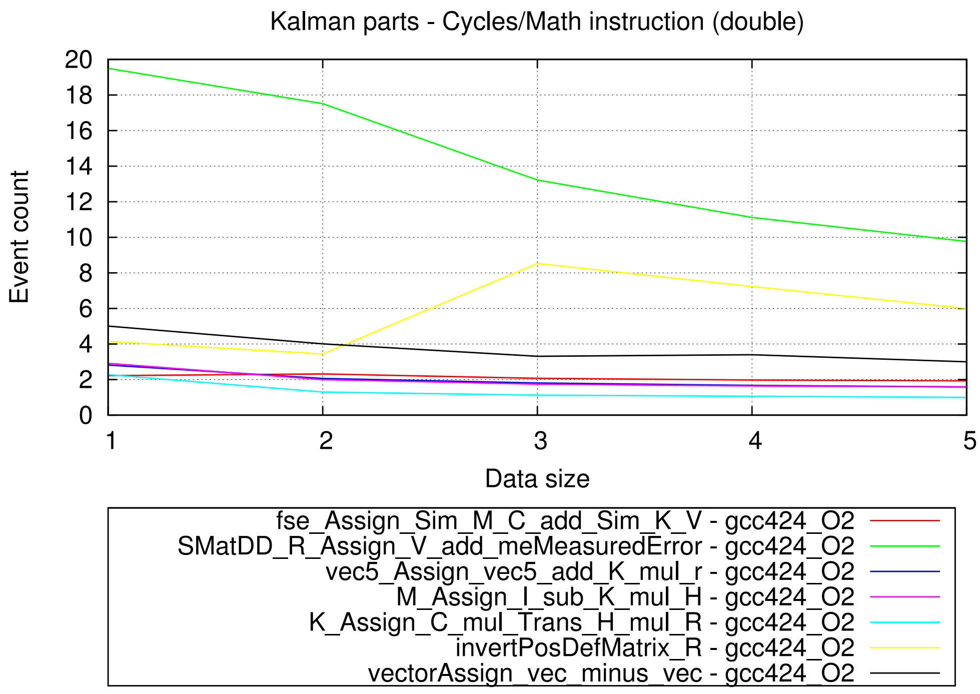

Figure 4.6: The Cycles pr. Math Instruction of different parts of a kalman filter.

An overview of the performance of the most expensive part (fse - Similarity) using different compilers can be used to select a better compiler version. Figure 4.7 shows the performance achieved by different compilers using only the O2 flag, indicating that GCC 4.4.0 might be a good candidate. Evaluating possible flag combinations for GCC 4.4.0, Figure 4.8 shows that unrolling improves performance and requesting code generation for Core 2 processors reduces performance. A look at what O3 might give is shown in Figure 4.9.

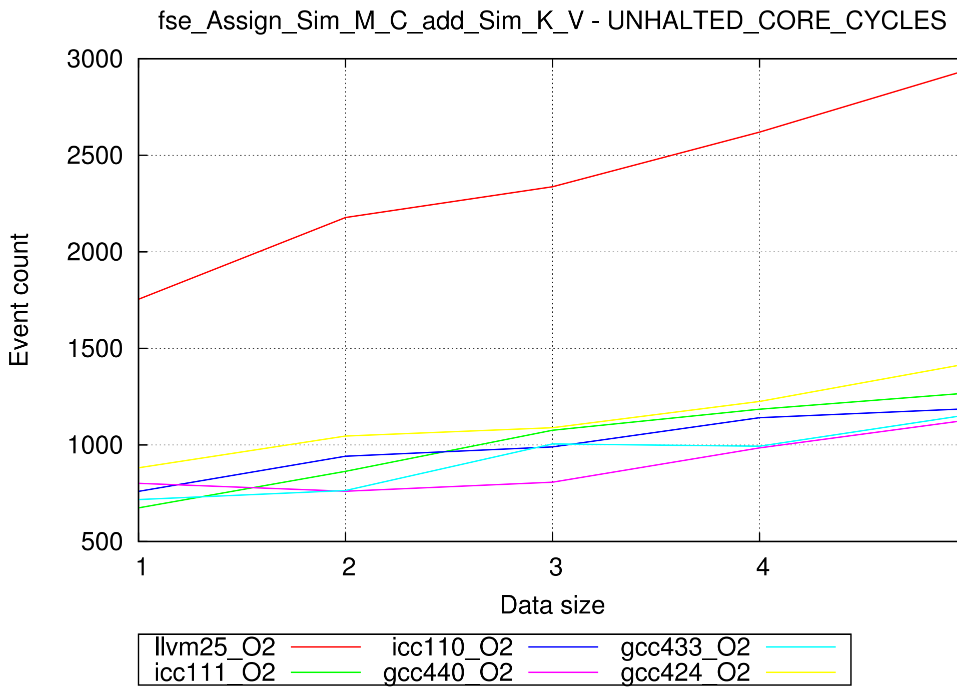

Figure 4.7: Cycles spent on similarity using O2 with several compilers.

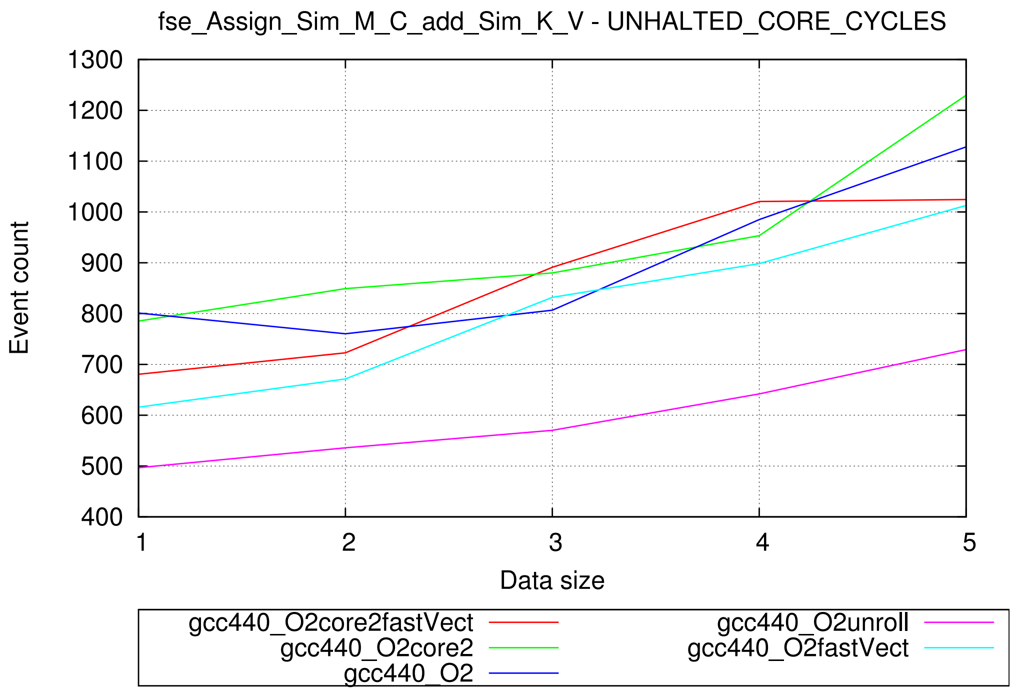

Figure 4.8: Performance using GCC 4.4.0 with O2 and a set of extra flags.

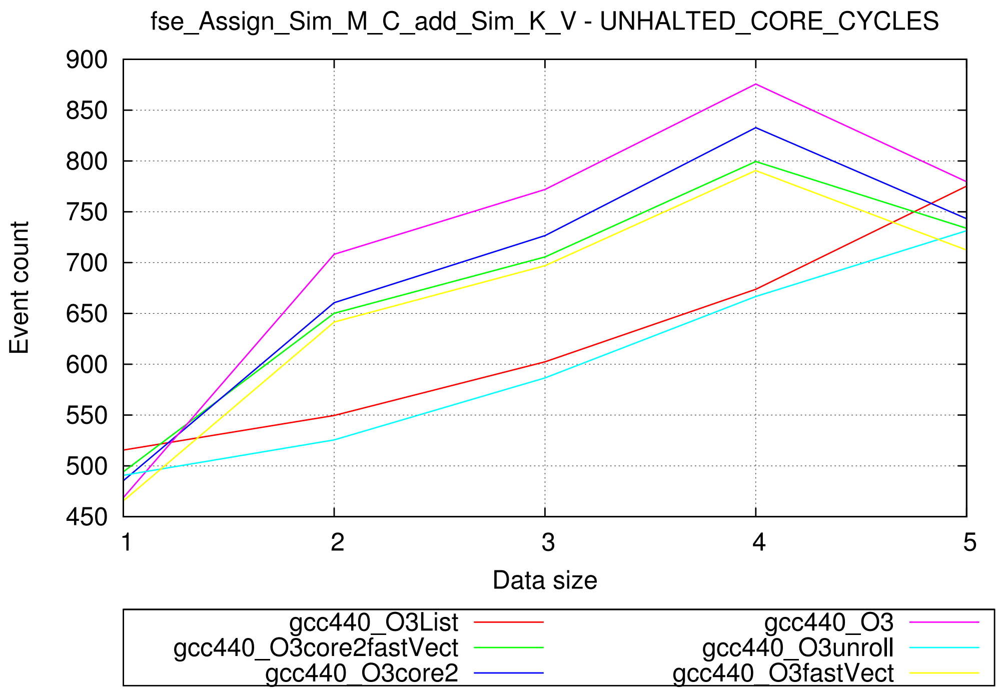

Figure 4.9: Performance of GCC 4.4.0 with O3 and a set of extra flags.

4.2 Performance Issues

Several issues which reduce performance have been identified. The most important is the non-linear runtime increase with long SMatrix expressions. A test program, *testExpressions.cxx*, demonstrating this behaviour was made. A somewhat modified version was also used as basis for a compiler bug report, where compilation time varied from 1 second to around 20 minutes. Three different ways of writing a chain of matrix multiplications are shown in Equations 4.1, 4.2 and 4.3 (from slowest to fastest), showing orders of magnitude difference in performance.

$$ANS = A * B * A * B * A * B \tag{4.1}$$

$$ANS = (A * B) * (A * B) * (A * B) \tag{4.2}$$

$$ANS = A * B; ANS* = A; ANS* = B; ANS* = A; ANS* = B; \tag{4.3}$$

4.3 Auto-Vectorization Evaluation

While both ICC and GCC are capable of auto-vectorization, they have limitations. One of the problems is that they often report successful auto-vectorization but no packed math instructions are generated. In order to evaluate the amount of packed math instructions generated Equation 4.4 has been used.

$$Packed\_Ratio = \frac{Packed\_math\_instructions * 2}{Packed\_math\_instructions * 2 + Scalar\_math\_instructions} \tag{4.4}$$

Figure 4.10 shows that the amount of packed math with the costly "fse" code is non existent for all versions and flags with GCC, except for an unneeded data size of 8. Looking at ICC, Figure 4.11 shows that plain O2 generates some degree of packed math instructions. Even if there are some packed math, the performance was not better than GCC (as seen in Figure 4.7).

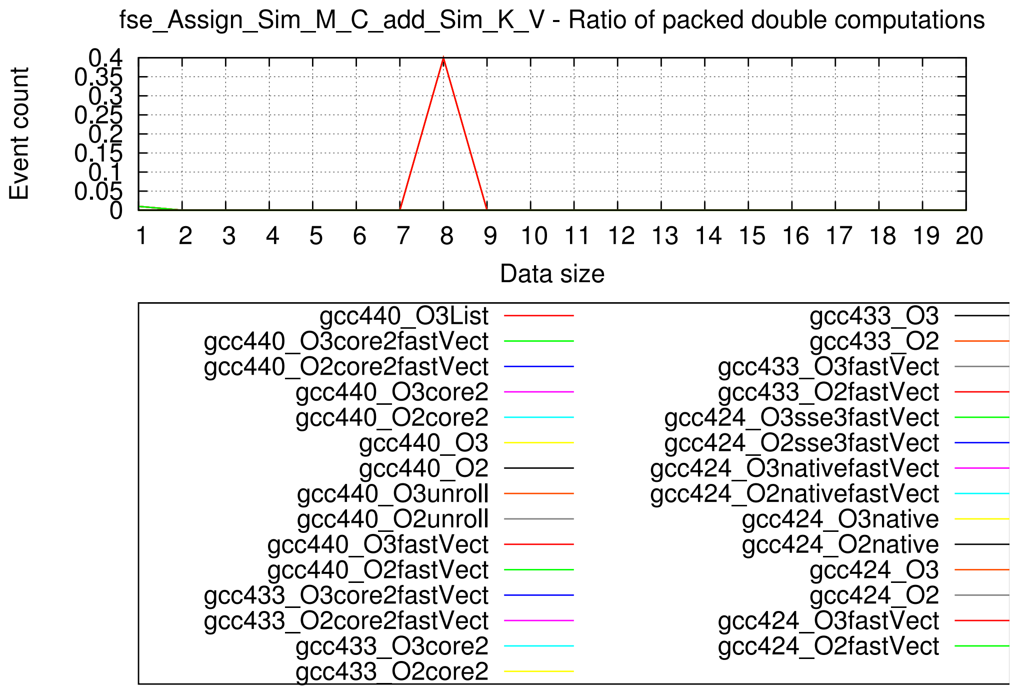

Figure 4.10: Fraction of packed math with several versions of GCC, each with several flags.

4.4 Optimizations

For several of the root test cases, both gcc and icc reported numerous successful auto-vectorizations. When using the feedback to track down the code being auto-vectorized most were data copy, constructors and other data initializations. Most computational code was either not considered or deemed too complex for analysis.

In order to help the compilers numerous changes were attempted. Some were simple and quite straight forward, while others were somewhat complex. Due to time constrains only a brief overview of some of the changes performed is explained in this report.

4.4.1 Simple Changes

Usage of *__attribute__ ((aligned (16)))*: the debug information from gcc indicates that this attribute was utilised. However, it was found not to be the reason auto-vectorization failed.

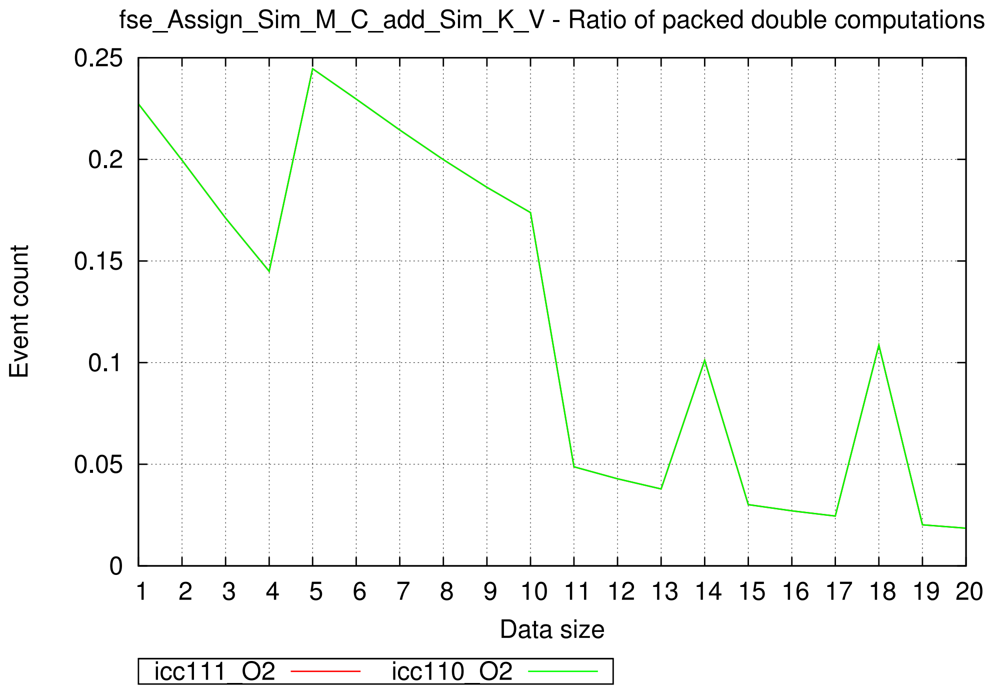

Figure 4.11: Fraction of packed math with two versions of ICC, using only the O2 flag.

4.4.2 Symmetrical Matrix Storage

When symmetrical matrices are used, SMatrix utilizes a packed storage layout to reduce the required storage space. In order to access elements in the packed storage a separate translation table is used, one for each matrix size. The translation table is created the first time a symmetrical matrix is used, requiring an *if* test in every function - testing if the table has been made. While the performance impact of the test itself is small, this design requires that all matrix accesses use one table access as well (unless the table is fully cached in registers). This indirect access pattern also prevents possible optimisations as the compiler cannot identify that almost half of the accesses use the same data. A number of other optimisation strategies might also fail, especially auto-vectorisation.

In order to prevent the runtime usage of the translation table, an optimized version was created. By pre-calculating the tables and making them both *const* and *static*, the access method could be improved. With the new design the compiler has access to the translation table at compile time, eliminating its usage at runtime. This reduces the number of memory references needed, removing a layer of indirection. This change was submitted to the author of SMatrix, its performance improvement verified, and will be included in the next release version.

4.5 Current and Future Work

From preliminary evaluations of SMatrix and SVector it seems that the possibilities for performance improvement are substantial. The main problem is to find ways to extract this potential, without major rewrites of the current template design. Several modifications were initially tested, but obtaining an accurate overview of their performance impact proved to be problematic. Solving these problems in a way that gives a broad enough picture, so that robust performance gains across all relevant compilers and data sizes could be verified, was not possible until after the framework in Section 3 had been completed. Because of time constraints this evaluation still remains to be finished. Only the symmetrical storage modification seemed to give a consistent improvement, and therefore remains the only contribution to SMatrix. Several other modifications gave only isolated improvements. Performing a broad comparison using cycle cost as the basis, it is possible to find and isolate changes that will improve the overall performance. By summing the cycle cost of each function or expression, corrected for the number of times they are used, a more unbiased performance scale can be designed. This approach will therefore (correctly) increase the importance of time consuming matrix operations, while the performance of simple vector operations will have little impact.

Chapter 5

Hand Optimization

Several functions where selected for hand optimization. This gives some insight to both the performance gain possible and the work needed to achieve it.

5.1 G4AffineTransform

Hand-optimized versions of the InverseProduct function in the G4AffineTransform class have been created using various methods. Several versions were made using enumerated data indexes, enabling changes to the data layout without rewriting the code preforming the calculations. No speedup was achieved using this approach, but it formed the foundation for an intrinsic version that was consequently developed. This version uses only packed instructions, cutting the number of math instructions almost in half (a few redundant operations are performed). Unfortunately, the straightforward intrinsic version gave only a relative small speedup. The reason for the small performance gain seemed to arise from the overall design in which all the memory loads are performed first followed by blocks of sequential calculation chains. This might therefore lead to two stall phases, first from the block of load instructions and second from data dependencies in the calculation. In order to test this, a version with some manually interleaved operations was made, showing some performance improvements.

Using the framework (Chapter 3) a quick evaluation was performed, indicating that there are bottlenecks related to partial data forwarding (see Section 3.3.1). This and related issues might therefore artificially limit the performance in the original benchmark design. A redesign of the benchmark according to the findings in Section 3.3.1 was not performed because of time constraints.

5.1.1 Optimization Method

A quick description of the method used to make intrinsic code will be described in the following section.

Data Layout

Make a struct that is a union between an array and the original values. The array is aligned, and provides a way to read and write data with vector SSE instructions. The original values maintains compatibility with old code, that use the named variables.

Data Access

The code reads all data into SSE registers first, then shuffles values around in the registers. Each SSE register hold a pair of variables. When two pairs of data variables are stored in two registers, a vector math operation is performed on the two pairs.

Calculation Phase

Key code has been constructed in phases:

- 1. Data layout in the struct was selected, so that sequential values could be calculated at the same time.

- 2. Modify the old code to update the array directly, so that every value is correct with new array layout. (Remember to verify!)

- 3. Load all values with SSE instructions, into a set of registers. Give all pair of variables the name of the two original variables, except for calculation temporaries. (It helps debugging)

- 4. Select two values to be calculated by vector instructions.

- 5. When a needed variable pair is found to be missing, construct if from the original loaded variables using shuffles (or any other data movement instruction). Note that some shuffle combinations are hard to create.

- 6. Perform vector math operation on the needed variable pair(s).

- 7. After all calculations are performed on a pair, store it to the answer array.

- 8. Comment out the old code for updating the two values.

- 9. Rerun code, check answer to be the same. (note: remember to initialize the calculation data with a full range of values, and NOT just 0 or 1's)

- 10. If more variables GOTO: 3

- 11. Done (with the first version).

- 12. Reorder the calculations, so that loads, adds, multiplications and shuffles are interleaved (and tight dependency chains are eliminated). This can be hard to do manually.

Bibliography

[1] Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, December 2008.