Slow Fourier Transforms on Fast Microprocessors

Bernd Pfrommer
Taku Tokuyasu

February 21, 1996

Contents

1 Introduction

While machine designs are continually changing in the pursuit of raw performance, the full benefit of these improvements cannot be reaped without detailed knowledge of the underlying computer architecture. By the same token, careful analysis of the performance of real applications can reveal limitations of a particular architectural design.

In this project we set out to explore the performance of a modern workstation, namely an IBM RS/6000 Model 590, on a scientific application to be described below. The POWER2 architecture of the Model 590 has a large number of functional units, typifying the "Brainiac" (as opposed to "Speed Demon") approach to achieving high performance. It offers a highly optimized superscalar design with aggressive support for exploiting instructional level parallelism. A key feature of the POWER2 processor is the addition of performance monitor hardware which allows detailed information to be recorded on the number of executed instructions, processor cycles, counts of cache and TLB misses, etc. This capability gave us valuable feedback on the low-level performance aspects of our code.

As described below, the performance bottleneck for our application is a one-dimensional fast Fourier transform (FFT), which runs at about half the peak speed of the processor, even using the highly tuned IBM proprietary ESSL [8] library code. This paper describes our attempts to both reach this speed in our own hand-built code, and to explain why the speed is so low.

2 Importance of the 1d FFT for our scientific application

Our scientific application is solving a Schroedinger type of equation in the context of solid state physics. It allows us to compute forces, energies, and equillibrium configurations of atoms inside crystals, i.e. we can compute material properties from first principles.

To tackle the problem, we use a quantum-mechanical formalism (density-functional theory) to explore the energetics of atoms in various configurations. It turns out that a quantitative treatment of the electronic states, especially the calculation of the total energy of the electrons in the crystal, is sufficient to explain a large number of experiments. Moreover, predictions about the stability of new materials can be made to guide experimental scientists in cases where the experiment is difficult or expensive to perform. We refer the interested reader to references [9] and [10], which provide more details on the formalism and its applications.

The predictive power of the formalism has led to support by the National Science Foundation. The solid state theory group at Berkeley currently has a yearly allocation of about 3000 Cray C90 hours, and a growing amount (currently 5000 hours) of CPU time on RISC based machines such as the SGI Power Challenge and the Convex Exemplar.

Our computational problem shares many of the features common to scientific applications, such as intensive use of floating point operations, matrix manipulations characterized by iterative loops which exhibit good data locality, etc. Specifically, extracting the electronic energies is equivalent computationally to diagonalizing large matrices. It is most convenient (and fastest) to switch between a real-space representation for the data and a "momentum" (i.e. Fourier transform) representation and back, over the course of this computation.

Since matrix multiplication routines scale as n^3, whereas the FFT scales as $n \log n$, where n is the number of points in our computational grid, the matrix aspect of our problem will dominate performance at large enough problem sizes. Since outstanding matrix routines have been developed, this implies that we can be assured of near peak performance in this limit. We have found, however,

that the FFT dominates for problem sizes of interest to us here ($n = 32^3$ to 64^3), and we expect this to remain true for sizes up to $n = 128^3$.

The 3d FFT is composed of a series of 1d ffts. We performed some preliminary timing experiments for $n = 32^3$, which showed that the performance of the 3d FFT was about 16% less than the what we measured for the single 1d FFT. This clearly identified the 1d FFT as the computational bottleneck, so we have concentrated our efforts on optimizing this part of our code.

Having isolated the performance bottleneck for our code, we turn to a description of the IBM architecture, before describing our attempts to optimize the FFT.

3 POWER2 Architecture.

The POWER2 chipset improves upon the earlier POWER architecture, with considerable enhancements in the ability to exploit floating point instruction level parallelism. The POWER2 architecture is made up of five basic units (see Figure 1:

- The instruction control unit (ICU) dispatches instructions and handles branches;
- The fixed point unit (FXU) decodes and executes all instructions except branches and floating point arithmetic;
- The floating point unit (FPU) decodes and executes floating point arithmetic instructions;
- The storage control unit (SCU) handles communication between the processors and external units such as main memory and I/O;
- The data cache unit (DCU) is a set of four units, controlling access to the data cache.

The FPU has two independent ALUs, each of which can do a fused multiply-add instruction per cycle, of the form $y = ax + b$. The FPU is thus capable of performing four floating point operations per cycle. Given the processor clock rate of 66.5 MHz, this yields a 266 MFLOPS peak performance rate. There are also two quadword (i.e. four doubleword) interfaces to the FPU which, with the addition of load and store quadword instructions, help ensure that the execution units are maximally utilized.

Figure 2 describes the data flow in the FPU. The FPU receives four instructions from the ICU and buffers them. After a predecode and register-renaming stage, it places the instructions into one of the three queues: arithmetic, load, or store. Each of these pipelines can perform two operations simultaneously. However, since the FXU does all address decoding with its two execution units, the number of load/store operations is limited to two per cycle. We also note here that, while the quadword load/store instructions are required in order to fully utilize the available processing power, they can only be used for adjacent floating point storage and register locations. The arithmetic pipeline is made up of a multiply and an add stage, which is fully utilized for a dependent pair of multiply and add operations, with a resulting two-cycle latency.

The data cache is a four-way set associative write-back cache, with a capacity of 256KB in 256B lines. It is dual ported, thus supporting two quadword load/store operations per cycle, and can have one miss outstanding before blocking. A third port is used for reload/storeback memory operations.

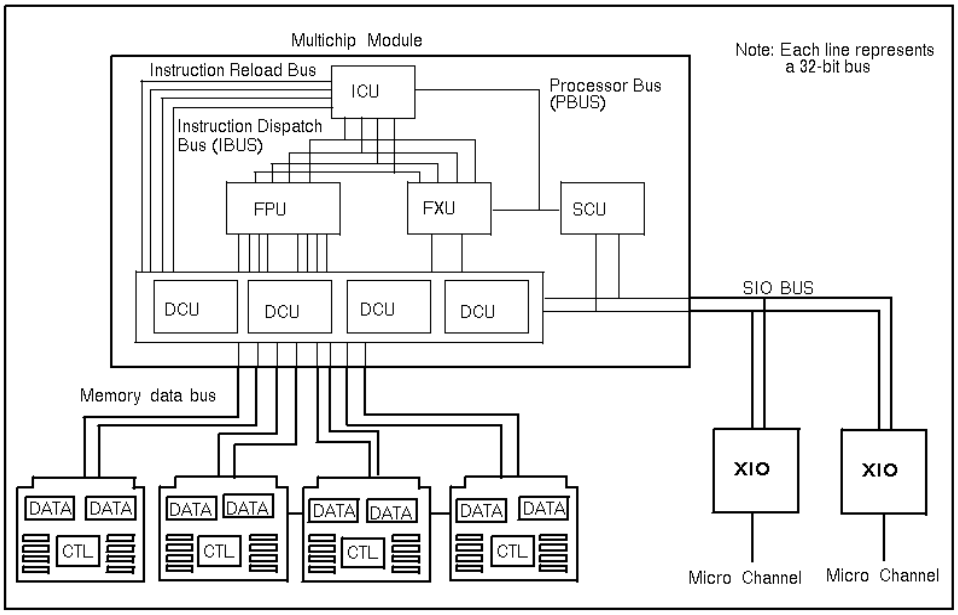

Figure 1: IBM POWER2 architecture

4 Performance monitor

The goal of the Rios 2 Performance Monitor is to provide detailed access to the behavior of the POWER2 chipset. Such information is becoming difficult to obtain, due to the increasingly integrated nature of processors, which reduces the number of signals which can be conveniently monitored off-chip. It accomplishes this with the help of twenty-two additional counters, which are physically located in the SCU. Five registers are allocated to count events in each of four units, i.e. the ICU, SCU, FXU, and FPU, and two registers are used as a cycle counter and soft error counter. A schematic overview of the monitor hardware [7] is given in Figure 3.

The five counters allocated to each unit cannot monitor an arbitrary set of events. Instead, each unit has sixteen possible event groupings defined for it. The actual set of events to be monitored is then determined by four 4-bit fields in the Monitor Mode Control Register (MMCR), which can be set in software.

For example, the events within the FPU which can be monitored include the number of floating point operations executed by each of the two units, the total number of cycles, and the number of hold cycles due to data anti-dependencies.

A number of library routines are provided to initialize the performance monitor, and to start and stop the counters. The counter values are accessible via an ordinary C struct data structure. We created a more convenient interface to these functions, and a typical output looks as follows:

```
——— performance monitor data ————-
totflopsadd: 0
totflopsmult: 0
totflopsfma: 24576
total number of flops: 49152
data dependencies: 0+0+0+0=0
```

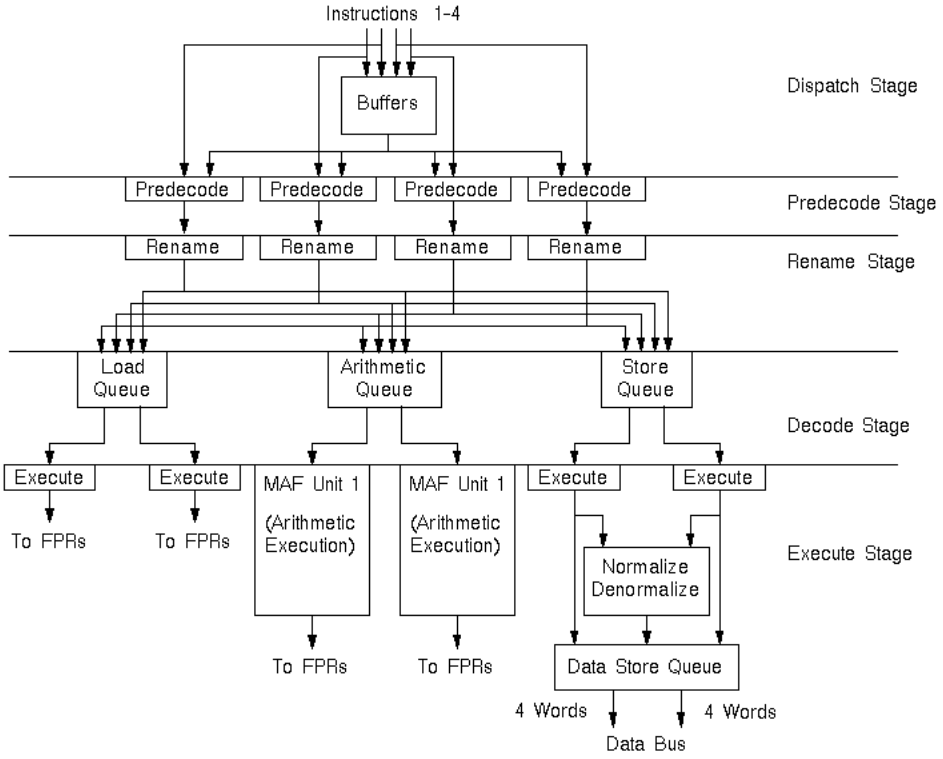

Figure 2: FPU dataflow

dcache misses: 0
hold cycles for dcache: 0
hold cycles fxu for fpu: 1539
hold cycles fxu for branch: 526
hold cycles fpu for anti: 1666
hold cycles fpu for load: 0
hold cycles fpu for store: 7682
hold cyc fpu validnotsunk: 0
hold cyc fpu validandsunk: 3332
cycles load queue full: 0
cycles store queue full: 11793
total number of cycles: 21742
total time: 0.000325
MFLOPS: 151.466502

Getting consistent results with the performance monitor proved to not be so simple. We noticed initially that for runs that lasted beyond about 100 milliseconds (ms), the cycle count began to vary significantly between runs. We got around this problem by timing the same code five hundred times and taking the minimum value. This produced numbers that varied by less than 0.1 percent over several runs of this type (subject to the caveats discussed below). The significance of the 100 ms time interval is somewhat mysterious. Although scheduling time quantum used by the operating system is quite a bit smaller (10 ms), we nevertheless feel this effect is related to context switches.

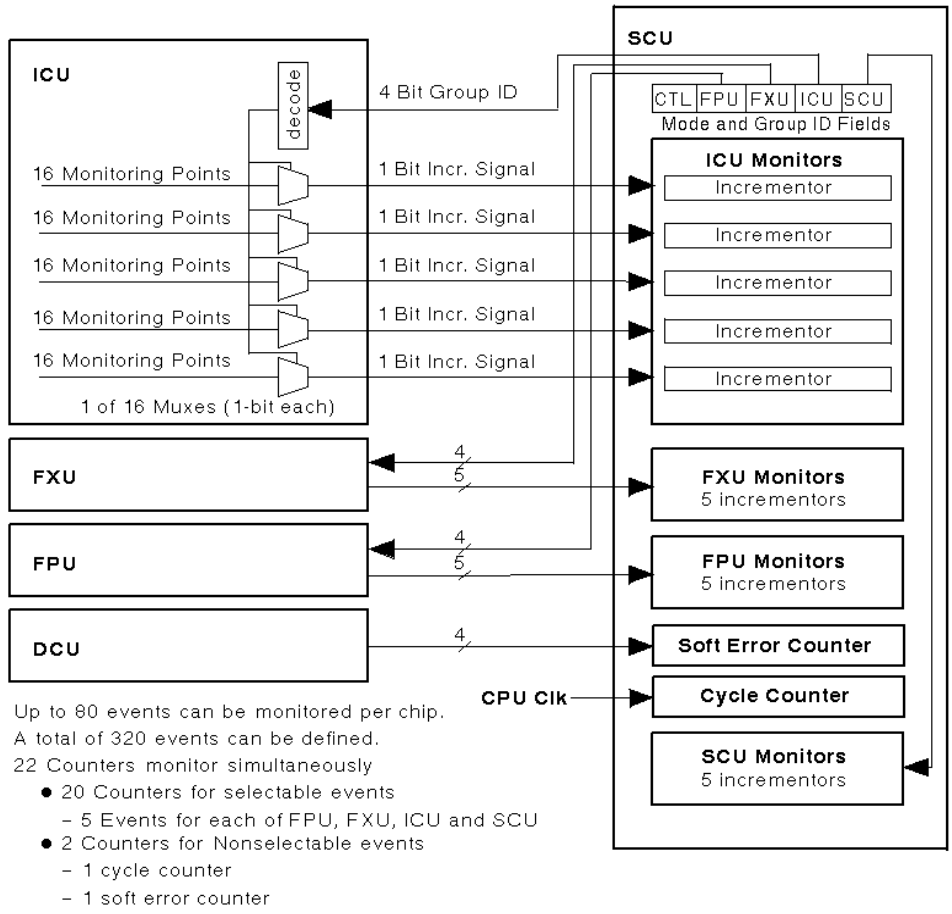

Figure 3: IBM Performance Monitor hardware

We recently noticed a more disquieting problem with the performance monitor. The cycle counts seem to depend on the environment variables set by the user! In particular, we have noticed that the cycle count can sometimes be reduced by ten percent simply by changing the terminal type from aixterm (the IBM terminal type) to hp (i.e. Hewlett-Packard). When running within a given environment, however, the performance numbers remain quite consistent. In addition, the cycle counts when running under the two terminal types can switch roles later in the day.

We are presently at a loss to explain this extraordinary behavior. It would be interesting to perform our experiments on a dedicated machine. As noted in Reference [6], clean timing results can often only be obtained after a reboot, for example, and we suspect that we may be in a similar situation. We leave this work to the future.

While these observations give us pause from assigning great precision to the performance monitor results, we also believe that they do reflect the actual performance of the machine, typically to within a percent. We thus continue with the analysis of the results, with the caveat that some of our conclusions will depend somewhat on the resolution of the above anomalies.

5 Algorithmic Prefetching

Assuming that an optimized 1d FFT is provided by the vendor, what can be done to improve the performance of the 3d FFT?

The implementors of the IBM ESSL library use the concept of algorithmic prefetching [14] for hiding cache effects of linear algebra subroutines. They perform dummy loads on data long before it is being used, to assure that it is in cache when needed.

We first thought that this concept could be applied between calls to the 1d FFTs to mask cache latencies. It turns out that this is not possible.

There are two cases where algorithmic prefetching is of benefit.

- If one of the fixed point units (which execute floating point loads and stores) is sufficient to keep both FPUs busy, the other one can be dispatched to prefetch data without any penalty.
- If the data can be assumed to be out of cache, and the memory-cache bandwidth determines the speed of the execution, one of the fixed point units can prefetch data to get full utilization of the bandwidth. The other fixed point unit serves the FPUs. The advantage of algorithmic prefetching in this case is that not both fixed point units stall until the cache miss is resolved (the IBM 590 cache supports hit under one miss).

In either case, this requires to have access to the innermost loop to build in the prefetching. We did not pursue this issue when exploring the performance of our 1d FFT, because we found that the fixed point units cannot keep the FPUs busy even without any prefetching instructions, and the benefit of prefetching should be limited.

6 Fast Fourier Transform Algorithms

6.1 The Concept of the Fast Fourier Transform

A fast Fourier transform is an algorithm for efficiently computing a Fourier sum:

$$F_k = \sum_{j=0}^{N-1} e^{2\pi i jk/N} f_j \qquad 0 \le k < N-1 \tag{1}$$

where f_j is the complex data in time space (or real space) at the N equidistant mesh points and F is the Fourier transform of f, computed at N points F_k in frequency space (or k-space). With the definition of the "twiddle factor":

$$W_N := e^{2\pi i/N} \tag{2}$$

the sum in (1) can be rewritten as:

$$F_k = \sum_{j=0}^{N-1} W_N^{jk} f_j \qquad 0 \le k < N-1 \tag{3}$$

Since the sum has to be performed for every $0 \le k < N-1$ and each sum has length N, the direct sum requires of order N^2 operations.

With a fast Fourier transform (FFT) algorithm, the sum can be performed with order $N \log_2(N)$ operations. It is based on the simple Danielson-Lanczos theorem (see for instance [11]). It states

that a Fourier sum of length N can be computed as the sum of two Fourier transforms of length $N/2$:

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j && (4)\\
&= \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i(2j+1)k/N} f_{2j+1} && (5)\\
&= \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/N} f_{2j} + W_N^k \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/N} f_{2j+1} && (6)\\
&= F_k^{even} + W_N^k F_k^{odd} && (7)
\end{aligned}
$$

In order to perform a Fourier transform of length N, one need to do two Fourier transforms F^{even} and F^{odd} of lengths $N/2$ on the even and odd elements, respectively. The two "subtransforms" can then be combined with the appropriate twiddle factors W^k to give the desired Fourier transform F. Note that in (7), the index k is restricted to the interval $0 \leq k < N$, because F_k^{even} and F_k^{odd} are periodic in k with length $N/2$.

Applying the Danielson-Lanczos theorem recursively leads to the Cooley-Tukey radix two FFT algorithm. In a radix four scheme, the Fourier transform would be partitioned into 4 subtransforms of length $N/4$.

6.2 The Cooley-Tukey radix two FFT algorithm

In this section, we outline the algorithmic aspects of the radix two FFT, which we ended up implementing.

6.2.1 Bit reversal

The recursive application of the Danielson-Lanczos theorem requires to order the data according to parity (even/odd) in a suitable fashion such that the successive stages of the FFT can be performed efficiently with a minimum of address arithmetic.

It can be shown [11] that if the bitpattern of the index to the datapoints is reversed, and the data is rearranged according to the so-obtained new indices, the bookkeeping of the FFT algorithm is simplified considerably.

This memory access intensive rearrangement phase is the first part of the FFT scheme. The computation of the rearrangement pattern can be done outside the main FFT routine, and stored in a workspace array, along with the twiddle factors (see section 6.2.2). This is of advantage if an FFT of the same length is computed several times.

6.2.2 Schematic outline of the implemented algorithm

The FFT algorithm consists of $\log_2$(N) stages, in which successively longer subtransforms are computed from the shorter length Fourier transforms obtained in the previous stages. The following example (see also figure 4) illustrates the scheme.

In the first stage of an FFT of length 8, the subtransforms are of length one (i.e. the data as it is) are used to compute four subtransforms of length two:

$$F_k = F_k^{even} + W_k F_k^{odd} \qquad\qquad 0 < k < 1; \tag{8}$$

More explicitly, for each of the four subtransforms one computes:

$$F_0 = F_0^{even} + (-1.0)^0 F_0^{odd} \tag{9}$$
$$F_1 = F_0^{even} + (-1.0)^1 F_0^{odd} \ . \tag{10}$$

In this particular case, the twiddle factor is $W_2 = -1.0$. We can see that it is of advantage to first load the twiddle factor into processor registers, and then always do F_0 and F_1 simultaneously, storing the two numbers right back into the places where F_0^{even} and F_0^{odd} have been before. This is allowed, because F_0^{even} and F_0^{odd} will not be used anymore during this stage.

Another nice feature of the twiddle factors allows the trick for the first stage to be used in higher stages also:

$$W_{N_{sub}}^{k+N_{sub}/2} = -W_{N_{sub}}^{k} \tag{11}$$

Thus we can do the following for all stages successively:

1. pick a certain $0 \leq k < N_{sub}/2$
2. compute (or load) the twiddle factor $W_{N_{sub}}^{k}$
3. loop through all the N/N_{sub} subtransforms and for each
   - compute *both*, F_k, and $F_{k+N_{sub}/2}$
   - write F_k and $F_{k+N_{sub}/2}$ back into the place where F_k^{even} and F_k^{odd} had been stored before.

After this has been done for all $0 \leq k < N_{sub}/2$, a stage has been completed: the scheme proceeds by doubling N_{sub}, and moving on to the next stage.

To repeat: One always computes *two* elements of the subtransform simultaneously such that they can be stored back into memory immediately. Also, the innermost loop runs over all the subtransforms to be done, since all those can be done with the same twiddle factor, and W can be kept in registers.

The following pseudo code illustrates the structure of the FFT, and is central to the understanding of our work.

```
for(Nsub=2;Nsub <=N; Nsub<<=1) /* loop over log2(N) stages, start with Nsub=2,
                                  double the size of Nsub every time */
 {
    for (k=0;k<Nsub/2;k+=1)    /* loops through 0<=k<Nsub/2. */
     {
       w = get_from_memory_twiddle_factor(Nsub,k);

       /* now do all the parts of the transforms which can be
          accomplished with the same  twiddle factor, e.g.
          the kth and the (k+Nsub/2)th element of subtransform 1,2,3...  */
```

```
      for (i=0;i<N/Nsub;i+=1)  /* loop through the different subtransforms,
                                  and always do k, k+Nsub/2 simultaneously */
        {

         /* compute temp = w * F_k_odd */

          temp.re =  w.re * data[i*Nsub+k+Nsub/2].re
                   - w.im * data[i*Nsub+k+Nsub/2].im;
          temp.im =  w.re * data[i*Nsub+k+Nsub/2].im
                   + w.im * data[i*Nsub+k+Nsub/2].re;

          /* now compute F_(k+Nsub/2)= F_k_even - temp   and store it */

          data[k+i*Nsub+Nsub/2].re=data[k+i*Nsub].re - temp.re;
          data[k+i*Nsub+Nsub/2].im=data[k+i*Nsub].im - temp.im;

          /* now compute F_k         = F_k_even + temp   and store it */

          data[k+i*Nsub].re += temp.re;
          data[k+i*Nsub].im += temp.im;
        }
    }

}
```

The twiddle factors can be computed beforehand in a separate subroutine, and reused if several FFTs of the same length are to be performed.

The rearrangement of the data in the bitreversal stage leads to the simple memory access pattern shown in figure 4. At every stage of the FFT, two subtransforms adjacent in memory are combined into a single subtransform of twice the length.

7 Efficiency of a fused multiply-add architecture for FFT algorithms

To achieve peak speed on the Power and Power2 architecture, we have to make efficient use of the fused multiply-add (fma) feature. Whenever a simple multiply or add is issued, half of the floating point unit's power is wasted, since it could complete an add or multiply in the same clock cycle. This is why we first focussed on this issue.

Let us have a look at the floating point operations happening inside the innermost loop of the pseudo code:

```
          temp.re =  w.re * data[i*Nsub+k+Nsub/2].re
                   - w.im * data[i*Nsub+k+Nsub/2].im;
          temp.im =  w.re * data[i*Nsub+k+Nsub/2].im
                   + w.im * data[i*Nsub+k+Nsub/2].re;
```

```
data[k+i*Nsub+Nsub/2].re=data[k+i*Nsub].re - temp.re;
data[k+i*Nsub+Nsub/2].im=data[k+i*Nsub].im - temp.im;

data[k+i*Nsub].re += temp.re;
data[k+i*Nsub].im += temp.im;
```

It contains a total of 10 floating point operations:

- 2 multiply operations
- 2 fused multiply-add operations, counting as 4 operations
- 4 add operations

The Power2 floating point unit [12] can perform two fma operations simultaneously, and would take 4 cycles to complete the above sequence of code (not counting the latencies). The fma feature is used only during one cycle. Ideally, the Power2 architecture can do 10 FLOPs in 2.5 cycles, i.e. a straightforward implementation of the simple arithmetic above is already doomed to get at most 63% of the peak speed (167 MFLOPS).

Work has been done on this matter, especially in the field of signal processing, and in the context of special purpose signal processing chips. Not much attention has been paid to implementation on RISC architectures. The most recent article we could find was reference [15] (1993), which reports a special kind of FFT algorithm and gives performance number for an implementaton on the power Architecture.

7.1 A simple FFT implementation suitable for a fused multiply-add architecture

When looking at reference [15] we realized that the implementation effort for their algorithm is considerable, especially if it had to be done in assembly language. Additional coefficients akin to the twiddle factors have to be precomputed and used during the computation.

However, the reward justifies the effort, since they reported to need only 3 instead of 4 cycles for the innermost loop of their radix two "scaled" FFT algorithm (leading coefficient in table II page 103 reference [15]).

To avoid the algorithmic complexity of the scaled FFT algorithm, we stared at the naive algorithm for a while, and came up with the following trick. Rather than computing (all operations are understood complex):

$$\begin{aligned} t &= wF^{odd} \\ F_k &= F^{even} - t \\ F_{k+N_{sub}/2} &= F^{even} + t \end{aligned}$$

we can do as well:

$$F_{k+N_{sub}/2} = F^{even} - wF^{odd} \tag{12}$$

$$F_k = 2F^{even} - F_{k+N_{sub}/2} \ . \tag{13}$$

That way, we do two excess multiply operations in multiplying by two in equation (13). However, now *all* operations are fmas, and the code sequence above runs in 3 cycles. The added programming

complexity is negligible, and as we will see, the different data dependency pattern does not pose any problems.

We have artificially raised the operation count of the FFT algorithm: Inside the innermost loop, we perform 12 instead of 10 FLOPs, so we have to correct our MFLOPS ratings by a factor of 5/6 to what we measure with the performance monitor.

The modified implementation of the FFT should be capable of reaching 5/6th of the peak speed, or 222.5 MFLOPS, assuming perfect scheduling of 2 fma instructions per cycle, and neglecting all other overhead.

There are other qualities of an FFT algorithm beyond pure speed. For many applications (and also our quantum mechanical application), a low accuracy Fourier transformation is sufficient. In other cases, roundoff errors introduced by the fourier transform might be undesirable. We have not studied the numerical noise possibly introduced by our changes to the algorithm.

7.2 How this relates to the Linzer/Feig paper

For completeness, we want to relate our little trick to the work of reference [15]. The understanding of the equation below requires to read reference [15], and *is not important to the understanding of the rest of the project.*

In the language of [15], equation (3), we propose a matrix factorization of the form:

$$\mathbf{A}(k,n) = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & -W_n^k \end{pmatrix} \tag{14}$$

The factor of 2 requires two additional multiplications (since we are dealing with complex numbers), but does not increase the cycle count.

8 Methodology

Although the IBM performance monitor was useful to measure overall cycle counts, and often provided valuable hints about performance bottlenecks, some quantities were not accessible. We used two different approaches to determine the performance.

- For determining the cycle count of the innermost loop, we could use the performance monitor, and we found a way of doing so while still measuring only the total cycle count of the FFT subroutine (see section 11.2).
- To measure the overhead for setting up the innermost loop, we devised a performance model. We adjusted the open parameters to fit the measured cycle counts, and could then read off the quantities of interest.

We had to build a performance model for two reasons.

1. The rather flexible renaming/scheduling scheme and the multiple functional units of the Power2 architecture makes it hard to predict interactions between different pieces of code, e.g. inner and outer loops. For example, leaving the innermost loop empty will not give an accurate measure of the loop overhead. We found that we could insert up to two FPU instructions inside the loop before noting any difference in the cycle count.

2. Short pieces of code cannot be measured accurately because of the interference of the performance monitor with the code.

We used the performance monitor to do simple tests on a series of isolated assembly code dummy loops to become familiar with the hardware. In those cases, the dummy loops did not interact with any other part of the code, and the counts from the performance monitor relatively easy to explain.

We focussed our efforts on a FFT of length $N = 1024$. The reason for picking this length were the following:

- It is a power of two (trivial).
- We wanted to compare to the performance of the well-optimized IBM ESSL library, which we timed to take about 22220 cycles, performing at 135 MFLOPS. The library routine has to do some sanity checks up front, check the integrity of the work arrays etc. We wanted this handicap to be negligible to get a somewhat fair comparison.
- To study the architectural aspects of the problem, we wanted to make sure that effects like filling and draining the pipelines do not distort the overall cycle count too much. This also facilitates the use of the performance monitor.

9 Expected performance for the Power2 architecture

What performance can we expect on the Power2 architecture for the innermost loop?

Let us look at the pseudo code, modified with our trick to get a better match with the fma architecture.

Assuming no scheduling problems, there are six fma operations to perform, which could happen in three cycles.

To feed the FPU units, we need to load two double complex variables, and store two double complex variables, provided the twiddle factor is kept in registers. The Power2 architecture has instructions to load and store quad words in one cycle. There are two busses to the dual-ported data cache, so 2 quad loads, or two quad stores, or one quad load and one quad store can be done during each clock cycle. A single quad load fetches a double complex variable, so during the three cycles while the FPUs are busy, one should be able to load and store three double complex variables. We only have two variables to load and store, and it seems like the FPUs should not stall on loads or stores.

Thus from just looking at the specifications of the Power2 architecture, we expect the innermost loop to run at full speed. We will find in section 12 that this is not the case.

10 Assembler versus C programming

The IBM ESSL library [13] is written in FORTRAN for portability reasons. We wrote the drafts of our routines in C, because we wanted to use explicit pointer arithmetic.

	no unrolling	2 times unrolled	4 times unrolled
xlc compiler	80.0	80.5	101.9
hand-tuned assembly	100.5	112.7	107.03

Table 1: MFLOP rating for a FFT of length 1024. The compilation of the C code was done with IBM xlc compiler and the options -O2 -qarch=pwr2

10.1 Compiling with different optimization levels

Ideally, we would like to have complete control at the source level about the arrangements of loops, and the exact sequence of statements in the assembly code. Switching off the optimization is a way of achieving those goals. However, on this level the compiler does not use the quad load/store instructions, and avoids using registers as much as possible.

Our experience is that working with optimization level -O3 is a bad idea, since the compiler does dramatic rearrangements of the source during the first pass. It is tedious to recognize the structure of the program from the assembly code, and the performance of the routine changes unpredictably when small modifications to the C source are done. When the statements in the C code are arranged such that they should lead to best performance, they were consistently rearranged to the worse. In such cases, compiling with level -O2 yielded faster code.

We used optimization level -O2 to generate a skeleton assembly source code. We then modified the assembly code to improve on the scheduling and the use of registers. To keep track of all the registers in the case of the four times unrolled loop, we wrote the innermost loop from scratch and used the GNU m4 preprocessor to merge it into the skeleton assembly source.

Table 1 gives a rough idea about the performance improvements we got by post-optimizing the assembly code. We discovered that the compiler did a reasonably good job with scheduling the instructions. However, it rarely uses all the available registers, and thus creates unnecessary antidependencies frequently. The quad load/store instructions load two adjacent registers simultaneously. This puts additional constraints on the use of registers. In several cases we found that the compiler could not handle this, and did not use the quad load/store feature.

In the case of the 4 times unrolled loop, there is more freedom for the compiler to manipulate the scheduling. In rescheduling, it also removed several stalls due to antidependencies, because instructions were moved inbetween, and the antidependencies did not lead to a stall.

11 Direct measurements using the IBM performance monitor

11.1 Measuring the bit reversal phase and other overhead

By removing and reinserting the bit reversal phase, we timed it to take about 5860 cycles. The overhead for the performance monitor and the call to the FFT subroutine was measured in the same fashion to be about 245 cycles. Thus the total overhead is around 6105 cycles.

11.2 Measurements of the innermost loops with the performance monitor

11.2.1 No Unrolling

Without unrolling the innermost loop, we expect stalls to happen due to data dependencies. The innermost part of the loop gets executed 512 times per stage, and has six fused multiply-add (fma)

instructions inside

After compiling the C source code, we first removed unnecessary anti-dependencies, and rearranged some of the scheduling.

To determine the performance of the innermost loop, we uncommented all loads and stores inside it. Then we replaced the floating point arithmetic fma instructions by other fma instructions, which we had tested before in a dummy loop, and which we know to run without stalls. Since there are six fma instructions inside the innermost loop, and two instructions can be scheduled in one cycle, we can assume a single loop iteration to take three cycles. We ran only the first stage, and measured a total count of 7531, including overhead and bit reversal.

Next, we put back the true fma instructions, and measured the cycle count for only running the first stage. It took 8551 cycles. Since the innermost loop is run 512 times, we infer that we encounter a 2 cycle penalty due to data dependencies between FPUs, i.e. a FP register could not become a target because it was the source of a previous instruction, which hasn't completed yet. The performance monitor indicated that there were 1536=3*512 cycles during which an FPU was waiting for either itself or the other FPU, indicating that during one cycle in the loop, both FPUs were holded, and another cycle, only one was stalled. Thus anti dependencies raise the cycle count by a factor of 1.67.

After restoring the loads and stores, we measured 9061 cycles for running the first stage, which indicated we had to take another stall cycle for the load/store operations. So rather than running at a speed of ideally 3 cycles, we measure the innermost loop to take 6 cycles. This way, by measuring the first stage of the FFT only, we determine the the innermost loop to run a factor of 2.0 longer than ideally necessary.

11.2.2 Two Times Unrolled Implementation

When the innermost loop is unrolled twice, there are 12 fma instructions inside the innermost loop, and it is executed a total of 256 times per stage. The task of unraveling the dependencies generated by the complier is tedious, but doable. We spent a lot of time to find what we are pretty sure is an optimal scheduling.

To determine the performance of the innermost loop, we proceeded the same way we did in the previous subsection. The ideal time for one loop iteration would be 6 cycles. Without the load and store instructions, we measure 6.5 cycles per iteration, or a slowdown of a factor 1.08. This is dramatically less than the factor of 1.67 encountered without unrolling the innermost loop. It indicates that every other loop iteration, the CPU has to stall for one cycle. The performance monitor counted around 256 FPU-FPU anti dependencies, which is consistent with a 0.5 cycle/iteration stall.

Putting back the load/store instructions costs another stall cycle, such that it now runs with 7.5 cycles/iteration, or a total slowdown factor of 1.25.

11.2.3 Four Times Unrolled Implementation

The data dependencies left in the case of the two times unrolled loop motivated the implementation of a four times unrolled loop. Three times unrolling is less advantageous because it is not a power of 2, and fringes have to be treated separately. The innermost loop now holds 24 fma instructions, and could run in 12 cycles/iteration.

We rewrote the compiler-generated innermost loop with the macro preprocessor to keep track of register usage and for ease of debugging. 28 out of the 32 available FP registers had to be used

to remove all data dependencies. We spent a lot of time until we had what we believe is the best scheduling that can be done.

Convinced that we should be able to get peak speed for the innermost loop, we tried scheduling for several days, just to find that we cannot go faster than in the two times unrolled implementation.

Anti dependencies are not a problem here, since the instructions are sufficiently far apart from each other, and neither are true dependencies. Without load and stores, we can get full speed, i.e 12 cycles/iteration. We can also insert all 8 quad loads loads and up to 6 of the 8 quad stores. But no matter where and how we scheduled the stores, more than 6 stores could not be put inside the loop without a performance hit. The best slots we could find raised the total cycle count from 12 to 15, resulting in the same slowdown factor of 1.25 as in the case of the two times unrolled loop.

The published literature [12] does not explain in full detail how the Power2 processor is organized. Especially the renaming scheme is not discussed sufficiently (for whatever reason). Based on the fact that the store queue (figure 8, reference [12]) has 6 stations, we conjecture that those 6 stations get filled, leading to a stall.

The performance monitor has a register to count the number of cycles the FP store queue is full, and we had many counts there. However, we had counts already with only 6 stores in place, indicating that the queue is full some times, but never stalls anything. We are interested in events where a full store queue leads to a stall. This could not be measured with the performance monitor.

Although we cannot pin down exactly what is wrong with the stores, we strongly assume that it is a scheduling problem, since the busses to and from memory have the capacity to handle the data transfer. Our best bet is the store queue problem, especially since we can schedule at most six stores per loop iteration, and the lenght of the queue is exactly six. It is also possible that the hardware runs out of internal registers, which we also consider a scheduling difficulty.

11.2.4 Summary on unrolling

We conclude that two times unrolling is sufficient, and that removing the data dependencies left there is not worthwhile doing, because then the store instructions limit the performance.

We find the two and four times unrolled loops to run a factor of 1.25 longer than ideal, and the version without unrolling to run a factor of 2.0 slower.

12 Determining the impact of loop overhead by fitting to the performance model

A peculiarity of the FFT algorithm is the loop count of innermost and second innermost loop. Consider the pseudo code, without unrolling. During the first stage, the innermost loop is called only once, but runs 512 times. During the second stage, it is called twice, and runs a length of 256 iterations. In the last stage, it is called 512 times, but iterates only once.

It is clear that during the last stage, the overhead for setting up the innermost loop, and loading the twiddle factor, will not be amortized on the single iteration during which the loop runs.

While the performance monitor is suited to measure the cycles spent inside the innermost loop, we cannot use it to measure the overhead for setting up the innermost loop. Firstly, removing or replacing instructions alters the behavior of the loop. Secondly, the processor leaves the innermost loop with state (e.g. the store queue is full), and thus the innermost loop interacts with the code surrounding it. As a matter of fact, the innermost loop runs one time less than its true cycle

count, and is preceded by a piece of code which preloads FP registers, and is followed by a code section which does a postcomputation. This part is interleaved with integer arithmetic instructions, and simply uncommenting the preload/postcompute code seriously alters the performance of the interleaved integer arithmetic instructions.

12.1 Performance Model

Let us have a look at the pseudo code presented in section 6.2.2. The length of the FFT is denoted N.

One part of the total time T_{tot} (all times are measured in clock cycles) is the time to call to the FFT subroutine, and to set up the outermost loop over N_{sub}. We shall lump this part of the total time into T_{overh}.

The cycle count T_{Bit} captures the time spent in the bit reversal phase.

Then we get a contribution of T_k for the section of the code which sets up the loop over k. It is executed N_s=$\log_2$(N) times, (the index s stand for "number of stages"), adding

$$T_k^{tot} = N_s T_k \tag{15}$$

to the total run time of the FFT.

There is also a cost for setting up the counters for the innermost loop. We expect this to be a rather important contribution, because during the later stages of the FFT, the innermost loop iterates only a few times, but is executed more often instead. We call the cycle count to set it up T_i. To find out how often it is executed, we have to evaluate a finite geometric sum, and arrive at a total contribution to the runtime of:

$$T_i^{tot} = T_i(2^{N_s} - 1) \ . \tag{16}$$

The section of code in the innermost loop executes for each stage $N/2$ times, since always two elements are processed in one loop iteration. If the loop is unrolled u times, the innermost part of the loop becomes u times longer, but executes only $1/u$ times as often. The computational kernal in the innermost loop thus contributes a total of

$$T_{kernal} = N_s N/(2u) T_1^{(u)} \ , \tag{17}$$

where $T_1^{(u)}$ is the runtime of one iteration of the innermost loop, which is unrolled u times.

Finally, the total runtime is

$$\begin{aligned} T_{tot} &= T_{overh} + T_{Bit} + T_k^{tot} + T_{kernal} + T_i^{tot} && (18) \\ &= T_{overh} + T_{Bit} + N_s T_k + N_s N/(2u) T_1^{(u)} + T_i(2^{N_s} - 1) && (19) \end{aligned}$$

At a first glance, it looks like we have sufficient parameters to fit a dog and its wagging tail. However, $T_1^{(u)}$ and T_{Bit} will be measured with the performance monitor, the only open parameters are T_i, T_k and T_{overh}. T_k^{tot} makes up a very small fraction of the total time, and is essentially unimportant for the performance of the algorithm. The same holds for T_{overh}.

To obtain the quantity T_i, we run the algorithm only up to a certain number of stages, i.e. we vary N_s, and measure the cycle count T_{tot} for different N_s. A least-square fit (done with Mathematica) will give us T_i from the fitting coefficients.

As a byproduct, we also get the sum $T_{overh} + T_{Bit}$ (the zeroth order terms) and the sum $T_k + N/(2u) T_1^{(u)}$ (the term linear in N_s). This allows us to compare with the numbers obtained from the direct performance monitor measurements.

stage	1	2	3	4	5	6	7	8	9	10
not unrolled	9061	11142	13775	16495	19221	21957	24719	27529	30625	34208
two times unrolled	8059	9672	11700	13723	15795	18020	20370	23105	26781	30302
four times unrolled	7981	10031	12072	14207	16436	18853	21646	25299	28467	32064

Table 2: Cumulative cycle count for different stages of the implementations of our radix two Cooley-Tukey FFT. The 10th stage of the two times unrolled version and the 9th and 10th stage of the four times unrolled version have to be considered separately.

	$T_{overh} + T_{Bit}$	T_1	T_i
not unrolled	6120(6105)	5.1 (6)	2.0
two times unrolled	5951(6105)	7.5 (7.5)	7.0
four times unrolled	5953(6105)	15.75 (15)	12.6

Table 3: Cycle counts for various quantities as derived from a fit to the performance model. The values in parenthesis show what we expected from our performance monitor measurements of the innermost loop and the overhead.

12.2 Fitting to the performance model

To collect the data for the performance model we ran our hand-optimized codes up to varying stages of the FFT, and measured the total cycle count with the performance monitor. Table 2 summarizes the cycle count data we collected for running up to stages one to ten.

Note that the last stage of the two times unrolled algorithm actually is only unrolled once. We had to write a separate section of code for this part, since the innermost loop runs only once, and hence cannot be unrolled two times. The same applies for the last two stages of the four times unrolled version: the second last stage runs on a two times unrolled loop, the last stage without unrolling. This has to be taken into account when fitting to the performance model.

A least square fit with Mathematica lead to the numbers shown in table 3. When doing the fit, we omitted the stages mentioned in the previous paragraph. The quantities we can read directly from the fit are T_i, $T_{overh} + T_{Bit}$ and $T_k + N/(2u)T_1^{(u)}$. Assuming that the time T_k to set up the loop over k is small compared to $N/(2u)T_1^{(u)}$, we computed T_1 to compare with the direct performance monitor measurements done in section 11.2.

The differences between the performance monitor measurements and the performance model are small, but remain unexplained. The assumption that T_k is small compared to $N/(2u)T_1^{(u)}$ is certainly valid, because $N/(2u)$ is at least 128. We note especially that $T_1 = 5.1$ instead of 6 in row one of table 3.

13 What we learn from the performance model

Graph 5 shows how well the fit tracks the measured data of table 2. We can see the mismatch for the last stages of the unrolled implementations: This is because for those stages the innermost loop cannot be unrolled, and a separate piece of code is executed. The two times unrolled version is the winner and has the lowest cycle count after stage 10, when the FFT is complete. We can also see how the 2^{N_s} function for setting up the innermost loop kicks in for the higher stages.

The two times unrolled version not only has the smallest slope, but also the time T_i to set up

the innermost loop is with 7 cycles considerably smaller than for the four times unrolled version, where T_i is 12.6 cycles. This pays off during later stages.

Note also the small time T_i=2.0 for the plain version, which shows as a more or less straight line in figure 5. The steep slope renders it a loser from the beginning, though.

The larger T_i for the unrolled versions can be understood as follows. The Power2 instruction set offers zero-cycle branches, and we use this feature in our assembly code. Furthermore, the quad load instructions automatically update the pointers, so striding through the data does not require additional integer instructions inside the innermost loop. For higher unroll factors, more pointers have to be initialized. In fact $2u$ pointers have to be set up (one for F_{even}, one for F_{odd}). The cost to initialize the loop is related to the number of pointers which have to be set up, which are four times more for the largest unrolling compared to the plain code.

We can also learn from figure 5 that it could be worthwhile implementing the later stages of the FFT with a smaller unrolling to avoid the increasing penalty of setting up the unrolled loop.

14 Summary of the performance analysis

The IBM 590 with its Power2 CPU running at 66.7 MHz has a peak performance of 267 MFLOPS. The fastest algorithm we could come up with is the two times unrolled radix two FFT, which delivers only 113 MFLOPS when computing an FFT of length 1024. Where do all the MFLOPS go?

Let us consider only the fastest, two times unrolled version of our algorithm. The total cycle count of 30302 can be broken down as follows.

- The bitreversal stage takes about 5860 cycles, which means a performance loss of about 19.3%, or 51.6 MFLOPS.
- Real work is only done inside the innermost loop, where 19200 cycles are being spent. But due to scheduling difficulties, there are three stall cycles per loop iteration, which then takes 7.5 instead of 6 cycles. That way, another 19200/7.5*1.5=3840 cycles, or 33.8 MFLOPS are lost.
- The time to set up the innermost loop is becoming increasingly important during the later stages of the FFT. It adds up to be a total of 30302-5860-19200=5242 cycles, leading to a 17.3% or 46 MFLOPS performance loss.

Because the FFT does not perfectly map onto the fused multiply-add architecture, some excess operations are done within the innermost loop. There are 6 fused multiply-add instructions required (instead of 5) to do effectively 10 FLOPs, thereby degrading the performance by another factor of 5/6, and losing another 22.6 MFLOPS. We are left with a 112.7 MFLOPS. Figure 6 summarizes where the MFLOPS go.

15 Conclusion

We studied the performance of an IBM POWER2 RS/6000 Model 590 on a 1d FFT. Through our experiments, we investigated the aspects of the architecture that are relevant for achieving high performance. Our conclusions are as follows:

- We found a new and simple way of using the fused multiply-add feature of the Power2 architecture for a radix two FFT. Although doing 20% excess arithmetic operations, we are able to write the algorithm in terms of *only* fused multiply-add instructions, resulting in 25% fewer arithmetic instructions than in the conventional algorithm. We relate our idea to the more complicated approach published in the literature [15].

- For our FFT, we find that the benefits of hand-optimizing at the assembly levels are substantial. This contrasts the approach reported in reference [13].

- Two times unrolling is enough for the innermost loop of a radix two FFT is sufficient.

- Unrolling more than two times actually decreases the performance, since the overhead for setting up the loop increases. This overhead becomes important during the last stages of the FFT.

- The innermost loop of the FFT algorithm has about 3 stalls out of 15 cycles, which is due to scheduling difficulties, presumably due to the short length of the store queue.

- Data accesses such as the bit reversal stage, and overheads for setting up the loop are intrinsic to the FFT algorithm, and reduce the pure floating point performance.

References

[1] Steven W. White, Sudhir Dhawan, POWER2: Next Generation of the RISC System/6000 Family, *http://www.austin.ibm.com/tech/POWER2.2.html#001*, Copyright IBM Corporation, 1994.

[2] http://www.austin.ibm.com/software/Apps/essl.html

[3] D.J.Shippy and T.W. Griffith, POWER2 fixed-point, data cache, and storage control units, *IBM J. Res. Develop.*, 38, 503, 1994.

[4] Troy N. Hicks, Richard E. Fry, and Paul E. Harvey, POWER2 Floating-Point Unit: Architecture and Implementation, *http://www.austin.ibm.com/tech/fpu.html*, Copyright IBM Corporation, 1994.

[5] Steven W. White, Sudhir Dhawan, POWER2: Next Generation of the RISC System/6000 Family, *http://www.austin.ibm.com/tech/POWER2.2.html#001*, Copyright IBM Corporation, 1994.

[6] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, second edition, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.

[7] E.H. Welbon, C.C. Chan-Nui, D.J. Shippy, and D.A. Hicks, POWER2 Performance Monitor, *http://www.austin.ibm.com/tech/monitor.html*, Copyright IBM Corporation, 1994.

[8] http://www.austin.ibm.com/software/Apps/essl.html

[9] P. Hohenberg, W. Kohn, Phs. Rev.**136** (1964) B864

[10] W.E. Picket, "Pseudopotential Methods in Condensed Matter Applications", Computer Physics Reports 9, 115-198 (1989)

[11] W. H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, Numerical Recipes, 2nd edition, Cambridge University Press, 1992

[12] T.N. Hicks, R.E. Fry, P.E. Harvey, IBM J. Res. Develop. 38, 525, (1994)

[13] R.C. Agarwal, F.G. Gustavson, M. Zubair, IBM J. Res. Develop. 38, 563, (1994)

[14] R.C. Agarwal, F.G. Gustavson, M. Zubair, IBM J. Res. Develop. 38, 265, (1994)

[15] E.N. Linzer, E. Feig, IEEE Transactions on Signal Processing, 41, 93 (1993)

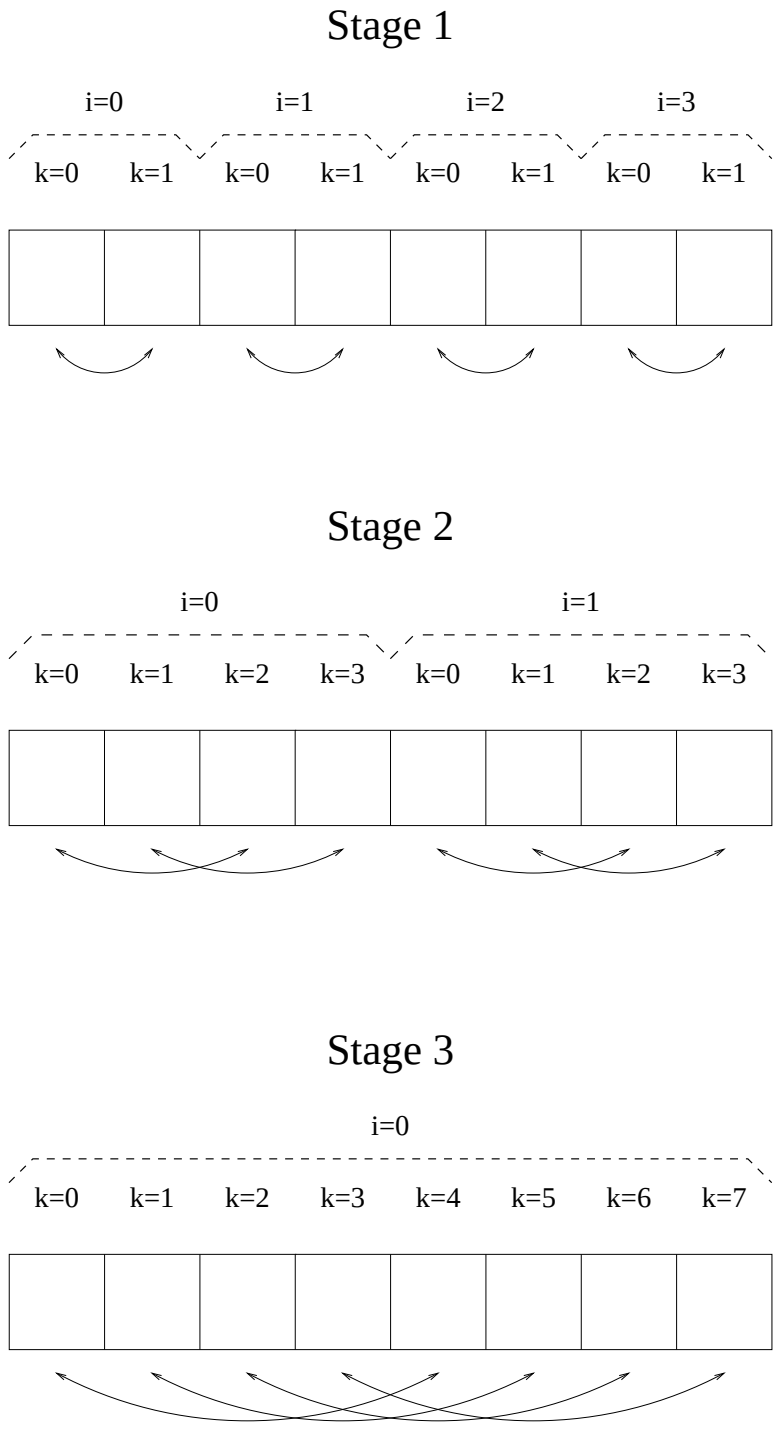

Figure 4: Memory access pattern of a radix 2 Cooley-Tukey algorithm. The length of the transform is $N = 8$, and there are $\log_2(N) = 3$ stages. Prior to the computation, the data has been rearranged in a bitreversed fashion. The variable names i (indicating the number of the subtransform) and k (the number of the processed element within each subtransform) correspond to those in the pseudo code. The arrows indicate the two elements which are computed together, and stored back into the memory locations of the source operands

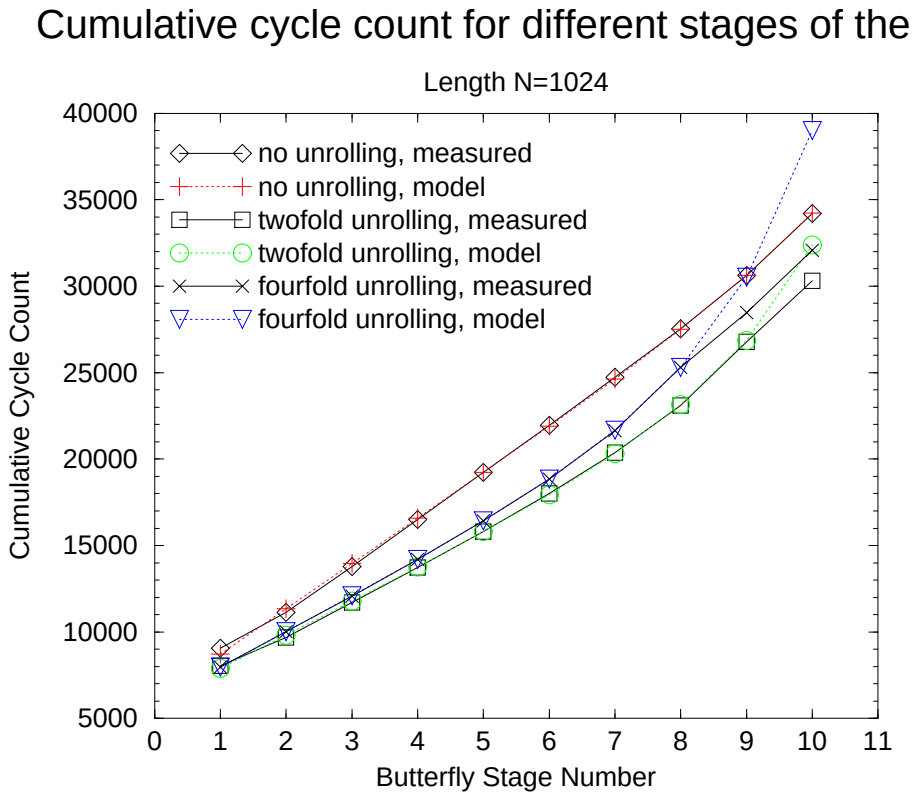

Figure 5: How the performance model tracks the measured data. We can see the mismatch for the last stages of the unrolled implementations: This is because for those stages the innermost loop cannot be unrolled, and a separate piece of code is executed. The two times unrolled version is the winner and has the lowest cycle count after stage 10, when the FFT is complete. We can see the 2^{N_s} function for setting up the innermost loop kick in for the higher stages.

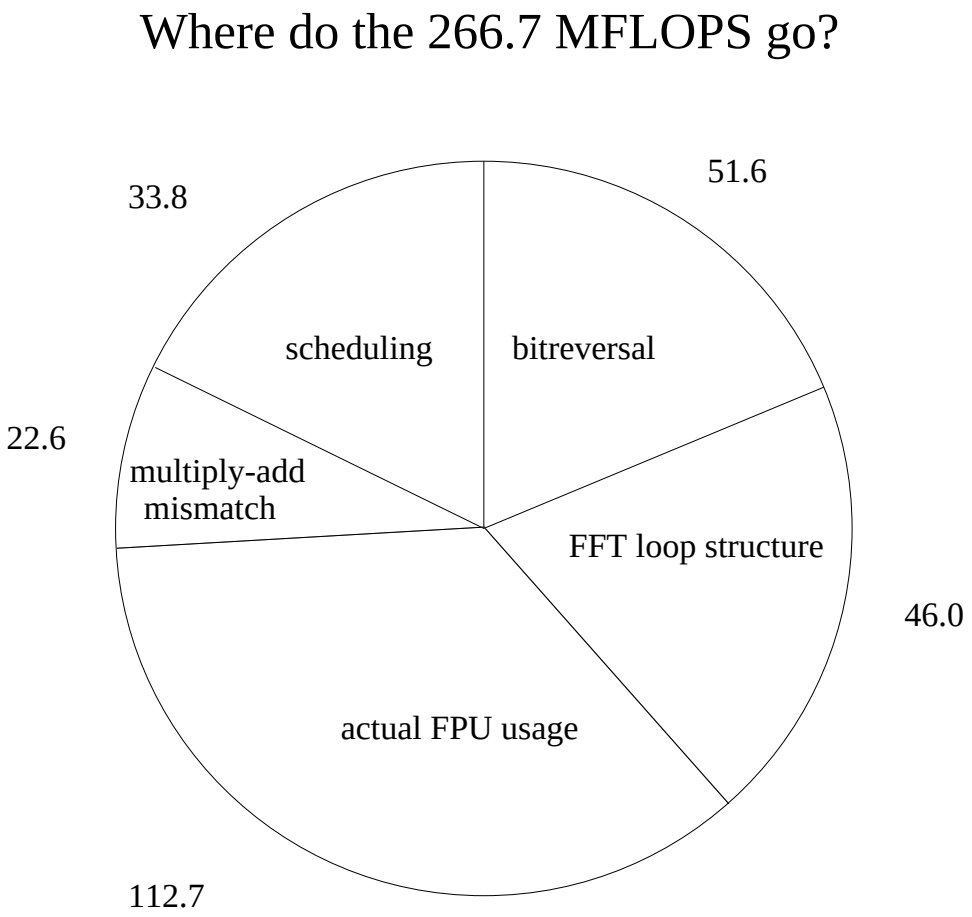

Figure 6: This is the breakdown for our two times unrolled radix two Cooley-Tukey algorithm. Ideally the entire pie would be devoted to FLOPs. The actual utilization is about 42%