Experience of Optimizing FFT on Intel Architectures

Daniel Orozco, Liping Xue, Murat Bolat, Xiaoming Li, Guang R. Gao
The Electrical and Computer Engineering Department
University of Delaware

Abstract

Automatic library generators, such as ATLAS [11], Spiral [8] and FFTW [2], are promising technologies to generate efficient code for different computer architectures. The library generators usually tune programs using two layers of optimizations: the search at the algorithm level, and the optimization for micro kernels. The micro optimizations are important for the performance of library, because the optimized micro kernels are the bases of algorithm level search, and have a great impact on the overall performance of the generated libraries. A successfully optimized micro kernel requires thorough understanding of the interaction between architectural features and highly optimized code. However, literature on library generators focus more on the algorithm level optimization, and usually give only simple discussion of how kernel codes are generated and tuned. As a result, the optimization of micro kernels is still an art that depends on individual expertise, and is insufficiently documented. In this paper, we study the problem of how to generate efficient FFT kernels. We apply a series of micro optimizations, for example, memory hierarchy locality enhancements, to several FFT routines, and use hardware counters to observe the interactions between those optimizations with Intel Pentium 4 and the latest Intel Core 2 architecture. We achieve good speedups, and more importantly, we present methods that can be used to generate high-performance FFT kernels on different architectures.

1 Introduction

Automatic library generators are software tools that automatically generate and tune high-performance code for different computer architectures. Well-known library generators, such as ATLAS, FFTW and Spiral, can generate versions of kernel routines that achieve the same performance or even outperform the code in the manually optimized commercial libraries. For example, ATLAS generates BLAS-3 numerical computation routines, and the kernel of which is an implementation of matrix-matrix multiplication that is selected from many automatically generated or manually written versions. FFTW and Spiral generate highly-tuned FFT routines, by searching in the space of mathematically equivalent implementations of a FFT formula.

The outstanding performance of the code generated by the automatic library generators derives from the capabilities of the generators that they can describe a space of different implementations for a kernel routine, and search the best performing version in that space. Usually library generators carry out two levels of optimizations. At the algorithm level, library generators produce different algorithms for the same function. At the lower kernel level, library generators use a set of kernels that are aggressively tuned for different architectures to implement the different algorithms. The performance of the generated code is decided by both the results of the algorithmic search and the quality of the kernel code. In particular, recent work [12, 7, 9] suggests that some of the lower-level optimizations, including the usage of machine-specific intrinsic or assembly code, prefetching, or SIMDization, play an important role in the good overall performance of library routines. However, publications and documentations of library generators usually put more focus on the algorithm level search, and lack the detailed description of how the high-performance kernels are generated or tuned. This is a missing link because the optimization for kernel code requires the detailed understanding of the interaction between programs, in particular aggressively optimized programs, and architectures. As a results of the lack of documentation, the state of art of low level optimization still highly depends on programmer's personal capabilities. We need a better understanding on how to systematically generate high-performance kernel code.

In this paper, we study how to write and optimize a high-performance FFT kernel, in particular a kernel like the *codelets* used in FFTW. The codelets are the most basic building blocks of FFTW. What FFTW searches are different combinations of codelets. FFTW applies a series of optimizations on codelets. However, the documentation of FFTW does not provide a detailed description of how codelets are generate and optimized. In particular, not much are described about the intuitions why certain optimizations

1-4244-0910-1/07/$20.00

are important, in other words, we do not know how to write codelets for new architectures, or codelets that are generally effective for many architectures. This paper describes in details how we apply a series of manual compiler like optimizations to several FFT implementations. Moreover, we use hardware counters to verify how effective the optimizations are in achieving their desired effects, such as reducing number of instructions or improving memory hierarchy performance.

The rest of this paper is organized as follows. Section 2 discusses briefly the organization of FFTW and how FFTW generates codelets. Section 3 describes our baseline implementation of FFT. Section 4 explains optimizations that we apply to FFT, and why we select those optimizations. Section 5 presents the performance results on two Intel platforms, and finally Section 6 presents our conclusion.

2 FFTW

2.1 How does FFT work?

In FFTW [2][3], the Fast Fourier transformations are computed by an executor, which is composed of several highly optimized C blocks, called *codelets* [2] in FFTW . During the runtime, a planner is used to search the best way to compose codelets. Basically, the planner executes different plans and measures their speed to find the best composition using a dynamic programming algorithm. Given the best plan, the executor can interpret this plan with small overhead. FFTW also provides a high level codelet generator to automatically generate codelets. In the following paragraphs, we will explain in detail how the executor, planner and codelets generator work.

The Executor is composed of many optimized code sequences called codelets. The executor calls appropriate codelets according to a *plan*. The executor uses a recursive divide-and-conquer algorithm to maximize the memory locality.

The Planner produces many plans for the given input size, and measures their execution time respectively. After that, it uses a dynamic programming algorithm to find the best plans for the given input size. The planner also collects information about the machine and stores the information in memory for later usage.

The Codelet Generator automatically generates optimized code sequences for smaller input sizes which are called codelets. The codelet generator, which is called *genfft* in FFTW, operates in four phases.

- In the first *creation* phase, genfft produces a directed acyclic graph (DAG) to represent the algorithm of FFT. Each node of the DAG is an operator, and the node's children represent the operands.

- In the second *simplifier* phase, genfft uses a rule-based simplifier to simplify the DAG. In this simplifier phase, it performs algebraic transformations and common-subexpression elimination, which are similar to the corresponding optimizations used in compilers. Besides that, it also applies several DFT specific optimizations. One simple but effective optimization is to make all the numeric constants positive and propagate the minus sign along the edges of the DAG.

- In the third *scheduler* phase, genfft produces a topological sort for the DAG in order to maximize register usage.

- In the last phase, genfft unparses the DAG, and outputs the C code implementation of codelets.

3 A First Implementation of FFT

A possible implementation for the FFT algorithm can use the Cooley Tukey approach of divide and conquer [5].

Of all the possibilities for divide and conquer, we focus on the divide by 2 algorithm. This approach greatly reduces the number of computations performed in order to calculate the Fourier Transform of our data.

3.1 Algorithm

Our algorithm takes the even elements of the input vector and copies them into an allocated array of memory, then a recursive call to the function is made to calculate the Fourier Transform of that half of the input set. The same is done with the odd positions of the input vector. Finally, both vectors are merged by multiplying the second vector result by the corresponding weights, as described in [5].

The code was written without any explicit optimizations.

3.2 First results

We compared our baseline implementation with the FFTW library. As can be seen on Figure 1, this simplistic approach performs far below from the results of FFTW. The code can be further optimized to provide a better performance.

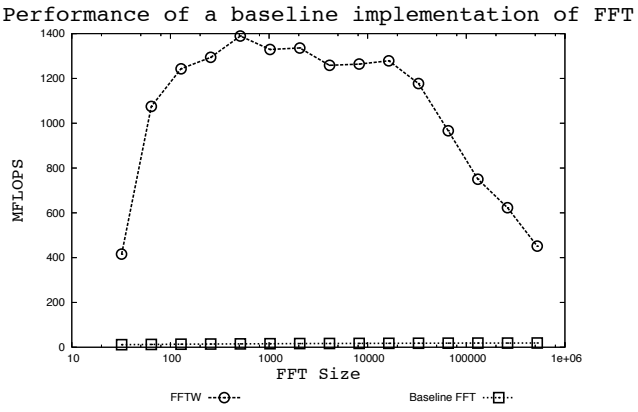

Figure 1: Comparison between a simple implementation of FFT and the commercial FFTW

4 List of Optimizations

4.1 Algorithm Design

The first optimization that a programmer can do is to carefully design the algorithm that will solve the problem at hand.

In the case of FFT, extensive research on algorithms has been done [10], [3], [8], [5] . The number of computations has also been calculated for each algorithm [4]. Since the purpose of this paper is to show how to manually optimize code at source code level, that is, without relying on assembly instructions, we selected the decimation-in-time Radix 2 algorithm as described in [4]. This algorithm does the same computations as the original implementation with the reentrant call, however, no memory copying is done, which by itself makes the computations run around twice as fast. The amount of memory required is of the order of N, the size of the problem, instead of $N \log_2(N)$ as in the previous version.

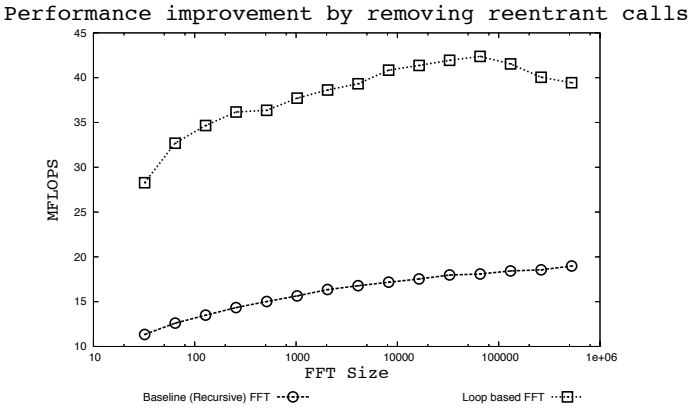

Figure 2: Comparison between a reentrant and a loop implementation of FFT

4.2 Removal of redundant computations

One of the major optimizations that can be done to a program is the elimination of redundant computations. Although today's compilers try apply this kind of optimization for many problems, identifying the redundance is a difficult task, in particular for redundancies at the algorithm level. The Radix 2 algorithm has redundant calculations that can be eliminated.

To name an example, in the FFT algorithm, at any of the $\log_2(N)$ computational steps,

$$X(k) = F_1(k) + W_m^k F_2(k) \tag{1}$$

where

$$W_m^k = \exp(-j2\pi\frac{k}{m}) \tag{2}$$

$$m = 2^s \tag{3}$$

$$s = \textit{FFT step} \tag{4}$$

$$F_i(k) = \textit{Frequency-domain sample} \tag{5}$$

It is easy to note that for k/m constant, the same calculation is done repeatedly for equation 2. Hence, for a size of $N = 8$, $k/m = 0$ for the pairs $(k, m) = (0, 2), (0, 4), (0, 8)$. This redundance greatly slows the program. In fact, the calculation of $\sin$ and $\cos$ takes most of the time in the original program.

A solution for this is to calculate at the beginning of the program the values of the weights, and store these values into an array. Since m changes at every step in the calculations, we do the computations for $m = N$, and whenever we need W_m^k we can use $W_N^{\frac{kN}{m}}$. Then, we can index this array to the appropriate position for the number that we want. It is true that the calculation of this index will take at least one integer multiplication, but nevertheless, it is still faster than the calculation of the trigonometric functions. Figure 3 shows the speedup of this simple optimization.

It is important to note that at each stage $N/2$ butterflies are computed. Each one of those will use some weights to calculate the result. Since the butterflies are independent for this step, the number of times that every W_m^k is required is exactly $N/2m$. This is yet another reason to support the precalculation of these values into an array.

The calculation of the position of W_m^k in the array of $W_N^{\frac{kN}{m}}$ can be done by taking the s least significant bits of k and then, multiplying them by N/m.

If the index is i, a very naive form of calculating it would be:

$$i = mod(k, m)\frac{N}{m} \tag{6}$$

This approach will require 2 integer divisions and one multiplication. There is a better way to do this,

$$i = AND(k, k_c) \cdot k_m \tag{7}$$

Where k_c is an integer with the lower s bits set to 1. The value of k_c can be calculated at the beginning of each stage and should not pose a significant overhead. The value of N/m can be also calculated just once per stage and stored into k_m.

There is still more room for improvement since the relationship between two consecutive values of k_m is just a factor of $1/2$, which can be achieved by a logical shift on the previous value of the variable, thus saving a division per stage. A similar approach can be used with k_c to calculate the next value. In other words, by doing

$k_c = (k_c << 1) + 1$
$k_m = (k_m >> 1)$

$>>$ and $<<$ being bitwise shift left and right operators, we can update the values of these variables. This improvements have shown an average improvement of 29% in performance.

Yet another redundant calculation that can be done is the reuse of indexes inside a loop. Usually, this is a job for the compiler, but on our case, the compiler failed to notice that in the following piece of code `2*(k^m)` is a constant.

```
out[2*k] =
sign*work[2*k] + work[2*(k^m) ];

out[2*k + 1] =
sign*work[2*k + 1] + work[2*(k^m) + 1];
```

Here, the compiler is not clever enough as to note that `2*(k^m)` is a constant in those instructions and does the calculation twice. By calculating that value, and storing it into a temporary variable, the program runs faster. It does however note that `2*k` is constant, and avoids calculating it several times.

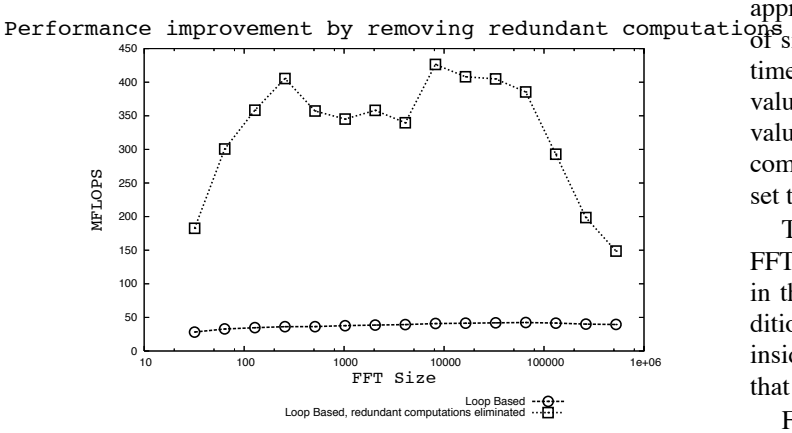

Figure 3: Improvement after removal of redundant computations

4.3 Mathematic improvement of the calculations

After applying optimizations in the section 4.2, we profile our implementation of FFT. From the profiling data, we can still see that the calculation of the weights takes a long time. The reason is that the $\sin$ and cos computations, which are essential steps in computing the weights, are very expensive. To reduce the time that is spent on the computation of weights, we note that

$$W_N^{k+1} = W_N^k \cdot W_N^1 \tag{8}$$

Also,

$$W_N^{k+\frac{N}{4}} = -jW_N^k \tag{9}$$

Note that each computation of W_N^k requires a call to $\sin$ and $\cos$ and at least a multiplication or an addition to get the angle argument for those functions. In our case, for $k < N/4$ (See equation 8) we need a complex multiplication, which is composed of 4 floating point multiplications and 2 floating point additions. For $k > N/4$, we can just use equation (9), that takes only sign changes . Since there is some error propagation due to the recursion, this can only be done for around 32 times in order to guarantee the 10^{-13} relative accuracy.

4.4 Bit reversal

One of the steps that must be done to calculate the Fourier Transform is to reorganize the sequence in bit reverse order. Since there are no single assembly instructions to achieve this on most processors, the programmer would have to do the bit reversal through explicit loops.

A slow approach to do a bit reversal would be to loop through all the numbers in the sequence and then, for each one of them, write a loop that will check for every bit in their address, and reverse it.

There are much better ways to do this. For example, an approach that proved very useful, was to allocate an array of size N, and then loop through the array $S = \log_2(N)$ times, using s as an iteration counter. At each loop step, the values of a single bit can be set for the s-th bit. Since these values are known in advance, instead of doing a conditional comparison to set or clear every bit, a loop can be written to set the bits in the positions needed.

This approach has shown to improve the performance of FFT. As long as we write tight loops, with few instructions in them, using non expensive instructions like integer additions and bitwise operators, and avoiding `if` statements inside the loops, the compiler will be able to schedule it so that few cycles are wasted in this step.

Further optimizations can be done for fixed sizes of N. A Look Up Table (LUT) can be used to get the precalculated answer for each address. This approach is faster, but without any improvements, the amount of memory required to hold the table would be too big to be a feasible solution for big sizes of N. Instead, by taking advantage of character indexing, 8 bits can be reversed at a time with a 256 position LUT. When the addresses have a length that is not a multiple of 8 bits, then, they have to be aligned by doing a shift operation before the LUT is used.

In Figure 4 two lines can be seen. The one labeled "Simple Loop" shows the number of cycles required to do the bit reversal of the addresses, per complex number, for a simple loop, that compares each bit and reverses it. The curve labeled "Using Array" shows the performance of the bit reversal when loops are used to set the bits in an array of addresses as explained before. A speedup of around two times in the bit reversal can be obtained by using the later algorithm

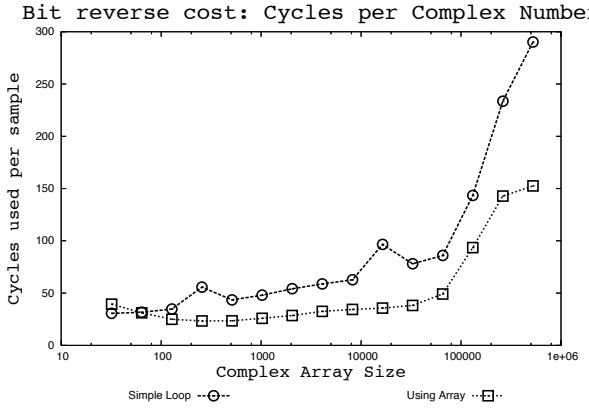

Figure 4: Comparison of the computational cost of bit reverse. Two implementations shown

4.5 Improving memory locality

If a program is carelessly written, the processor may spend a lot of time waiting for memory reads. In general, a good practice is to try to keep the data used by the program into cache memory.

The way numbers are stored in memory is reflected in the performance of the program. For example, if the real and the imaginary parts of the weights are stored in different arrays, the cache would have to fetch 2 lines of memory for every reference to a complex number. If we use the widely known convention of placing the real and imaginary part of a complex number close together, a speedup of around 10% can be seen.

It is also important to note that if extra instructions are needed to achieve memory locality, a performance improvement is not guaranteed. In particular, if more conditional instructions have to be inserted into the code, and the branch prediction mechanism is unable to predict them, the program can get slower. For example, a butterfly calculates 2 numbers out of 2 numbers:

$$y_i = x_i + x_j \quad y_j = x_i - x_j \tag{10}$$

In this case, if a loop is written to go sequentially over all the values of y_k, $k = 0, ..., n$, it would be faster than calculating the values of y_i and y_j at the same time while checking that they have not been calculated before. This is due to the fact that additional branch instructions are introduced into the code, and if the branch prediction system fails to predict them, the code will run much slower.

In general, the branch prediction system works very well for loops. So, for scientific computation over large sets of data, it can be said that a loop instruction will run faster than an `if - else` statement.

4.6 Avoiding Branches

In modern processors, the execution pipeline holds several tens of instructions. A mispredicted branch will waste several clock cycles.

One way to improve the performance of a program is to reduce the number of branches and conditional instructions. This not only lowers the number of instructions, but also avoids some of the branch misprediction scenarios.

Branch removal can be done with excellent results on tight loops. On these cases loop unrolling can be used.

By doing loop unrolling, p times, the number of branches can be reduced by a factor of p. In our case, by unrolling 4 times, we got a speed improvement of 25%.

Another good technique is to change a combination of a `for` loop with an `if` statement inside it for two `for` loops. This can not be done always, but it helps.

To name a case, on the 6th stage of the computations, (For an FFT of size 1024), the working vector has to be multiplied by the corresponding weights. Not all weights are multiplied, however. Only those who have their 6th bit set must be multiplied. One way to do this is by using an and operator with the hex constant $0x40$ which equals 64 in the following way:

```
for ( n = 0; n < 1024; n++ )
if ( ( n & 64 ) ) { ...  }
```

The loop can be changed to

```
for ( n = 64; n < 1024; n+=64 )
{
limit = n + 64;
for ( i = n; i < limit; i++ )
...  }
```

The second version runs faster, and executes less instructions. It only targets the specific elements that need to be computed, and the if statement is not used.

4.7 Kernels

In many situations, some calculations must be done over a relatively small amount of data. A good way to improve the performance is to write special code that solves the small computations. In the case of FFT, the computation of the first butterflies can be done using specially optimized code.

If this approach is used instead of writing standard loops for it, the performance of the overall program improves.

This step can be done at the same time that the memory is being bit - reversed. This way, loop iterations are saved, and the programmer can write instructions specifically intended to fully utilize the superscalar capabilities of the processor at hand.

To illustrate this, consider that at some point, the program calculates all the reversed addresses of the vector and stores them into an array called `bitr`, as described in the

bit reversal section. Also, lets define `in` as the input vector, `inr` as an array of 4 complex numbers (doubles, 8 positions), `y` as an array of 2 complex numbers, `i` (a multiple of 4) used as an index to `in`, and `out` as our final array where the kernel is computed.

The first two stages of the computations can done by doing:

```
inr[ j ] = in[ bitr[ i + j ] ]
```

Placing one instruction for each $j = 0, ..., 7$ then,

```
y[ 0 ] = inr[ 0 ] + inr[ 2 ];
y[ 1 ] = inr[ 1 ] + inr[ 3 ];
y[ 2 ] = inr[ 4 ] + inr[ 6 ];
y[ 3 ] = inr[ 5 ] + inr[ 7 ];

out[ 0 ] = y[ 0 ] + y[ 2 ];
out[ 1 ] = y[ 1 ] + y[ 3 ];
out[ 4 ] = y[ 0 ] - y[ 2 ];
out[ 5 ] = y[ 1 ] - y[ 3 ];

y[ 0 ] = inr[ 0 ] - inr[ 2 ];
y[ 1 ] = inr[ 1 ] - inr[ 3 ];
y[ 2 ] = inr[ 4 ] - inr[ 6 ];
y[ 3 ] = inr[ 5 ] - inr[ 7 ];

out[ 2 ] = y[ 0 ] + y[ 3 ];
out[ 3 ] = y[ 1 ] - y[ 2 ];
out[ 6 ] = y[ 0 ] - y[ 3 ];
out[ 7 ] = y[ 1 ] + y[ 2 ];
```

This piece of code exhibits excellent performance, since it is simple enough for the compiler as to schedule it appropriately, also, the compiler will probably be able to do a good register allocation so that time optimal software pipelining can be achieved [6]. Also, there are no close dependencies on the loop, allowing the compiler to fully pipeline the instructions in the floating point unit. Furthermore, the addresses have been calculated beforehand, so offsets can be used when this code is turned into assembly instructions, which optimizes the time spent calculating addresses.

Figure 5 shows the performance of our implementation of FFT for a kernel size of 4 and several sizes of N. This later implementation includes all the optimizations described before. Our implementation of FFT calculates the weights and the bit reverse addresses. This accounts for part of the slowdown when compared with FFTW.

5 Test Results

The algorithms are evaluated in two different architectures and for three different sizes of N. The evaluation sizes for N are 512, 1024 and 65536. Their results are compared to the results of *FFTW3* [3]. The tested architectures are Intel Core 2 Duo and Intel Pentium 4.

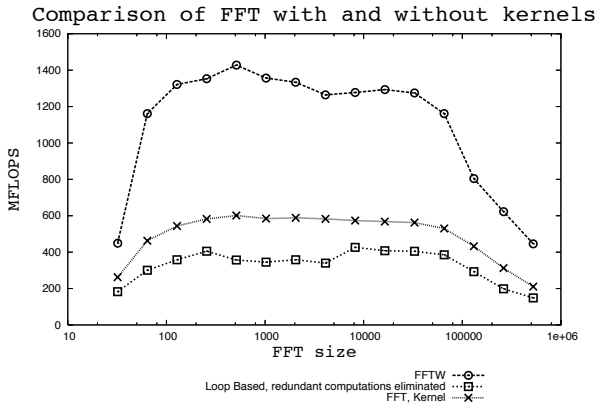

Figure 5: Performance improvement by using a kernel of size 4 against no kernels. FFTW uses bigger kernels an is shown for comparison

The Intel Core 2 Duo machine has following characteristics:

- Processors: 2 Intel Core 2 Duo Processor
- Frequency: 1596 MHz
- L2 Cache Size: 4 MB
- RAM: 4 GB

The Intel Pentium 4 machine has following characteristics:

- Processor: Intel Pentium 4
- Frequency: 3.05 GHz
- L2 Cache Size: 512 kB
- RAM: 512 MB

We used the ***PAPI*** [1] library to evaluate the code. We instrumented our algorithms and the *FFTW* implementation with different performance counters available in each machine. The performance counters which we use in Intel Core 2 Duo architecture are shown in Figure 6 and the counters which we use in the Intel Pentium 4 architecture are shown in Figure 7.

The results for the Intel Core 2 Duo Architecture are shown in Figures 8, 9, 10, The scale of the figures 8, 9 and 10 are logarithmic. The performance counters are shown without the prefix "$PAPI\_$ and performance counters which have zero for all versions are not shown in the figures.

The results for the Intel Pentium 4 Architecture are shown in Figures 11, 12, 13. The scale of the figures 11, 12 and 13 are logarithmic. The performance counters are

PAPI_BR_CN	Conditional branch instructions
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BTAC_M	Branch target address cache misses
PAPI_FDV_INS	Floating point divide instructions
PAPI_FML_INS	Floating point multiply instructions
PAPI_FP_OPS	Floating point operations
PAPI_TOT_CYC	Total cycles
PAPI_TOT_INS	Instructions completed
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_RES_STL	Cycles stalled on any resource
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_BR_INS	Branch instructions

Figure 6: Performance counters used in instrumenting Intel Core 2 Duo Architecture

PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_FP_OPS	Floating point operations
PAPI_TOT_CYC	Total cycles
PAPI_TOT_INS	Instructions completed
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_RES_STL	Cycles stalled on any resource
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_BR_INS	Branch instructions
PAPI_L2_TCM	Level 2 cache misses
PAPI_L2_TCH	Level 2 total cache hits
PAPI_L2_TCA	Level 2 total cache accesses

Figure 7: Performance counters used in instrumenting Intel Pentium 4 Architecture

shown without the prefix "$PAPI\_$ and performance counters which have zero for all versions are not shown in the figures.

The labels in the figures are:

- Version 1: This is our baseline implementation. It uses recursion to take advantage of the Cooley Tukey algorithm.
- Version 2: Here the recursion was changed for a loop based algorithm. No other optimizations were done.
- Version 3: Redundant computations were eliminated from the code of Version 2.
- Version 4: The algorithm was optimized to get better memory locality. All previous optimizations are also present here.
- FFTW: For comparison, the results of the FFT transformations done with FFTW3 are shown. FFTW3 uses less resources, in part due to the fact that this library has precomputed some of the steps required to do a FFT.

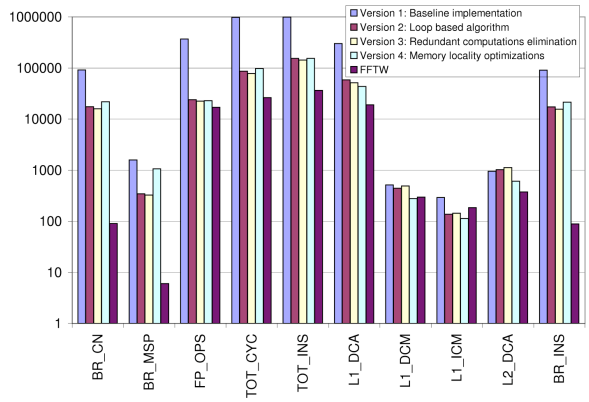

Figure 8: Measurement results on Intel Core 2 Duo for size $N = 512$

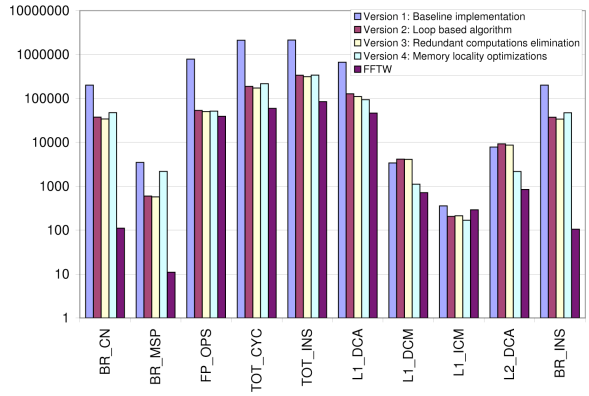

Figure 9: Measurement results on Intel Core 2 Duo for size $N = 1024$

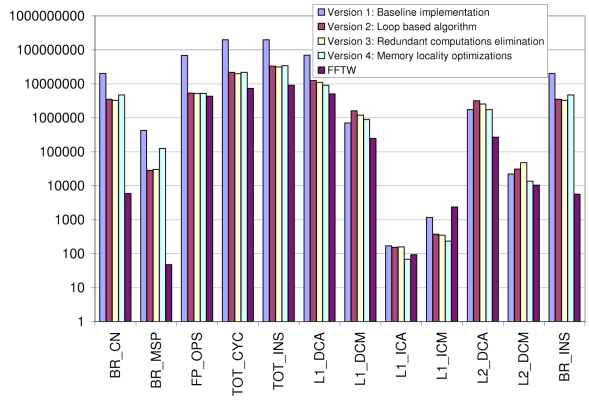

Figure 10: Measurement results on Intel Core 2 Duo for size $N = 65536$

6 Conclusions

In this paper, we study the problem of how to generate and tune high-performance FFT kernels. We try to answer the question through the manual optimizations of several FFT routines, and the close observations, using hardware

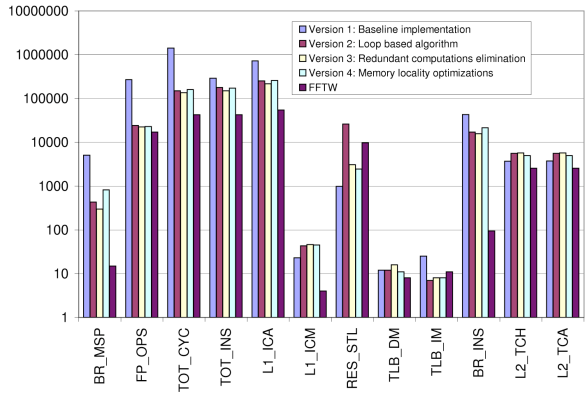

Figure 11: Measurement results on Intel Pentium 4 for size $N = 512$

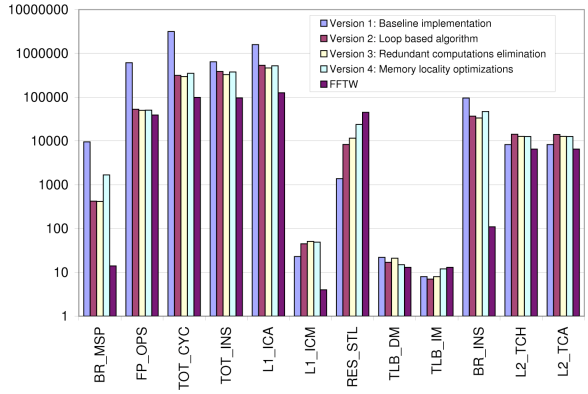

Figure 12: Measurement results on Intel Pentium 4 for size $N = 1024$

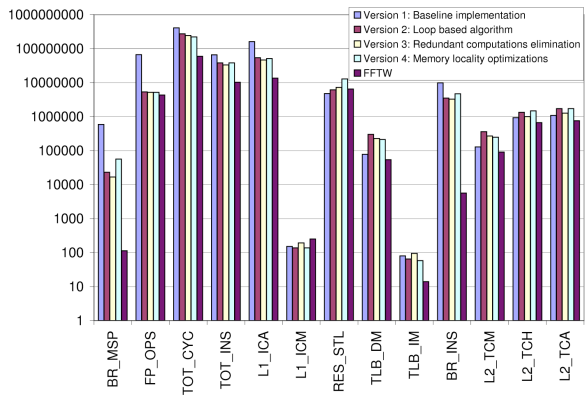

Figure 13: Measurement results on Intel Pentium 4 for size $N = 65536$

counters, of how those optimizations interact with two Intel architectures. We achieved good speedups comparing the tuned FFT kernels with our baseline implementations. More importantly, we revealed what optimizations are most effective, and how exactly they accelerate FFT routines, with extensive experiments using hardware counters.

This paper presents the first step towards our goal to understand how different levels of optimizations interact in automatic library generators. The next step of our study along the direction will be the research on how differently optimized kernels affect the algorithm level search, and how it changes the final outputs of library generators.

7 Acknowledgements

We thank the anonymous reviewers for their many detailed and constructive suggestions for improving this paper.

References

[1] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *sc*, 00:42, 2000.

[2] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.

[3] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[4] D. M. J Proakis. *Digital Signal Processing. Principles, Algorithms, and Applications*. Prentice Hall, 3 edition, 2004.

[5] J. T. JW Cooley. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 1965.

[6] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42, New York, NY, USA, 1993. ACM Press.

[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[8] M. e. a. Puschel. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *The International Journal of High Performance Computing Applications*, pages 21–45, Spring 2004.

[9] J. Shin, M. Hall, and J. Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *6th Workshop on Media and Streaming Processors (MSP6)*, June 2004.

[10] B. SORENSEN, HEIDEMAN. On computing the split-radix fft. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-34:152–156, February 1986.

[11] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[12] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas. *Proceedings of the IEEE, 93(2), 2005. special issue on Program Generation, Optimization, and Adaptation.*, 2005.