Chapter 5

Results

This chapter is organized as follows: First, we introduce our measurement methodology (5.1). Then, we present results that validate our overall optimizations in the context the StreamIt compiler (5.2). Finally, we present experimental results that provide additional justification of our methods: the effect of combination on performance (5.3); the effect of using ATLAS (5.4); the effect of problem size scaling on the FIR benchmark (5.5); the effect of removing redundancy (5.6); the effect of problem size scaling on the Radar benchmark (5.7); and the effect of implementation choices on the frequency transformation (5.8).

5.1 Measurement Methodology

The StreamIt compiler currently has two code generation backends. The uniprocessor backend generates sequential C code that is compiled and linked against a supporting library. The second backend generates code for the Raw microprocessor [35, 24], which features a grid of processors interconnected via various communication structures. We chose to use the uniprocessor backend for our measurements to control for the variability inherent in mapping a StreamIt program to the Raw architecture. This mapping is complicated by issues such as communication, load balancing and partitioning [15]. It is difficult to separate the effects of our optimizations from the effects of the different tile placement, routing, and fusion which result from modifying the program's structure.

The most appropriate performance metric is always debatable. In this thesis, we choose to measure the strength of our optimizations in terms of execution time and floating point instruction counts.

Execution Time Execution time is a complex function of the operating environment, complicated cache dependencies, and the super scalar architectures of modern processors. Additionally, the uniprocessor backend for the StreamIt compiler is meant for prototyping and the supporting runtime library is not optimized for speed. Improvements in uniprocessor running time *do* provide an indication of the potential real world benefits of our optimizations, but they should be interpreted with the above factors in mind.

Operation Reduction Our optimizations reduce the fundamental amount of computation required by our benchmark programs. To capture this reduction, we present the changes in floating point operation count alongside timing information. Our optimizations most commonly remove multiplications, so in the rest of this chapter we also present reductions in multiplications. In this thesis, our measurement platform uses the IA-32 instruction set. We define FLOPS as in Table 5.1[1]. We define multiplication instructions to be any of `fmul fmulp fimulp fdiv fdivp fidivp fdivr fdivrp fidivr`.

Our measurement platform is a dual 2.2 GHz Intel P4 Xenon system with 2GB of memory running GNU/Linux. We compile our benchmarks using StreamIt's uniprocessor backend and generate executables from the resulting C files using `gcc -O2`. To measure the number of floating point operations, we use an instruction counting DynamoRIO[1] client. We normalize both execution time and instruction counts to the number of program outputs generated. Since StreamIt is a new language, there are no external sources of benchmarks. Thus, we have assembled the following set of representative streaming components and have rewritten them in StreamIt:

[1]By Bill Thies from `http://courses.ece.uiuc.edu/ece291/resources/instructionref.pdf`

Flop?	Opcode	Description
▷	FABS	Absolute Value
▷	FADD	Add
▷	FADDP	Add
▷	FIADD	Add
	FBLD	Load Binary Coded Decimal
	FBSTP	Store BCD Integer and Pop
▷	FCHS	Change Sign
	FCLEX	Clear Exceptions
	FNCLEX	Clear Exceptions
	FCMOVcc	Floating-Point Conditional Move
▷	FCOM	Compare Floating Point Values
▷	FCOMP	Compare Floating Point Values
▷	FCOMPP	Compare Floating Point Values
▷	FCOMI	Compare Floating Point Values and Set EFLAGS
▷	FCOMIP	Values and Set EFLAGS
▷	FUCOMI	Values and Set EFLAGS
▷	FUCOMIP	Values and Set EFLAGS
▷	FCOS	Cosine
	FDECSTP	Decrement Stack-Top Pointer
▷	FDIV	Divide
▷	FDIVP	Divide
▷	FIDIV	Divide
▷	FDIVR	Reverse Divide
▷	FDIVRP	Reverse Divide
▷	FIDIVR	Reverse Divide
	FFREE	Free Floating-Point Register
▷	FICOM	Compare Integer
▷	FICOMP	Compare Integer
	FILD	Load Integer
	FINCSTP	Increment Stack-Top Pointer
	FINIT	Initialize Floating-Point Unit
	FNINIT	Initialize Floating-Point Unit
	FIST	Store Integer
	FISTP	Store Integer
	FLD	Load Floating Point Value
	FLD1	Load Constant
	FLDL2T	Load Constant
	FLDL2E	Load Constant
	FLDPI	Load Constant
	FLDLG2	Load Constant
	FLDLN2	Load Constant
	FLDZ	Load Constant
	FLDCW	Load x87 FPU Control Word
	FLDENV	Load x87 FPU Environment
▷	FMUL	Multiply
▷	FMULP	Multiply
▷	FIMUL	Multiply
	FNOP	No Operation
▷	FPATAN	Partial Arctangent
▷	FPREM	Partial Remainder
▷	FPREM1	Partial Remainder
▷	FPTAN	Partial Tangent
▷	FRNDINT	Round to Integer
	FRSTOR	Restore x87 FPU State
	FSAVE	Store x87 FPU State
	FNSAVE	Store x87 FPU State
▷	FSCALE	Scale
▷	FSIN	Sine
▷	FSINCOS	Sine and Cosine
▷	FSQRT	Square Root
	FST	Store Floating Point Value
	FSTP	Store Floating Point Value
	FSTCW	Store x87 FPU Control Word
	FNSTCW	Store x87 FPU Control Word
	FSTENV	Store x87 FPU Environment
	FNSTENV	Store x87 FPU Environment
	FSTSW	Store x87 FPU Status Word
	FNSTSW	Store x87 FPU Status Word
▷	FSUB	Subtract
▷	FSUBP	Subtract
▷	FISUB	Subtract
▷	FSUBR	Reverse Subtract
▷	FSUBRP	Reverse Subtract
▷	FISUBR	Reverse Subtract
▷	FTST	TEST (compares with 0.0)
▷	FUCOM	Unordered Compare Floating Point Values
▷	FUCOMP	Unordered Compare Floating Point Values
▷	FUCOMPP	Unordered Compare Floating Point Values
	FWAIT	Wait
▷	FXAM	Examine
	FXCH	Exchange Register Contents
	FXRSTOR	Restore x87 FPU, MMX, SSE, and SSE2 State
	FXSAVE	Save x87 FPU, MMX, SSE, and SSE2 State
▷	FXTRACT	Extract Exponent and Significand
▷	FYL2X	Compute y * log2x
▷	FYL2XP1	Compute y * log2(x +1)

Table 5.1: Intel IA-32 FLOPS.

FIR a single 256 coefficient low pass FIR filter.

RateConvert an audio down sampler that converts the sampling rate by a non-integral factor ($\frac{2}{3}$).

TargetDetect four matched filters in parallel with threshold target detection.

FMRadio an FM software radio with equalizer.

Radar the core functionality in modern radar signal processors, based on a system from the Polymorphic Computing Architecture [23].

FilterBank a multi-rate signal decomposition processing block common in communications and image processing.

Vocoder a channel voice coder, commonly used for speech analysis and compression.

Oversampler a 16x oversampler, a function found in many audio CD players.

DToA an audio post-processing stage prior to a 1-bit D/A converter with an oversampler and a first order noise shaper.

Table 5.2 shows characteristics of our benchmarks both before and after our optimizations. The stream graphs are presented in Appendix B.

5.2 Overall Performance

We have completed a fully automatic implementation of the linear combination, frequency replacement, and optimization selection algorithms described in the previous chapter. In this section, we evaluate three configurations of linear optimizations for the uniprocessor backend:

- *Linear replacement*, which transforms maximal linear sections of the stream graph into a single linear node, which we implement as a matrix multiply. For small nodes (less than 256 operations), this takes the form of an unrolled arithmetic expression, whereas for large nodes we implement an indexed matrix multiply that avoids zero entries at the top and bottom of each column.

- *Frequency replacement*, which transforms maximal linear sections of the stream graph into a single node which performs the calculation in the frequency domain.

	Originally			
Benchmark	**Filters (linear)**	**Pipelines (linear)**	**SplitJoins (linear)**	**Average vector size**
FIR	3 (1)	1(0)	0 (0)	256
RateConvert	5 (3)	2 (1)	0 (0)	102
TargetDetect	10 (4)	6 (0)	1 (0)	300
FMRadio	26 (22)	11 (9)	2 (2)	40
Radar	76 (60)	17 (0)	2 (0)	4412
FilterBank	27 (24)	17 (9)	4 (4)	52
Vocoder	17 (13)	10 (8)	2 (1)	60
Oversampler	10 (8)	1 (1)	0 (0)	33
DToA	14 (10)	3 (1)	0 (0)	52

	After Optimizations		
Benchmark	**Filters**	**Pipelines**	**SplitJoins**
FIR	3	1	0
RateConvert	4	1	0
TargetDetect	7	1	1
FMRadio	5	1	0
Radar	38	17	2
FilterBank	3	1	0
Vocoder	6	2	1
Oversampler	3	1	0
DToA	7	2	0

Table 5.2: Characteristics of benchmarks before and after running automatic selection optimizations.

To implement the necessary basis conversions, we use FFTW [6, 7, 8], which is an adaptive and high-performance FFT library.

- *Automatic selection*, which employs both of the previous transformations judiciously in order to obtain the maximal benefit. We use the algorithm presented in Section 4.3.

One interesting aspect of our optimizations is that they eliminate floating point operations (FLOPS) from the program, as shown in Figure 5-1. Figure 5-2 shows that multiplies are removed in roughly the same proportion. The removal of FLOPS represents fundamental computation savings that is independent of the streaming runtime

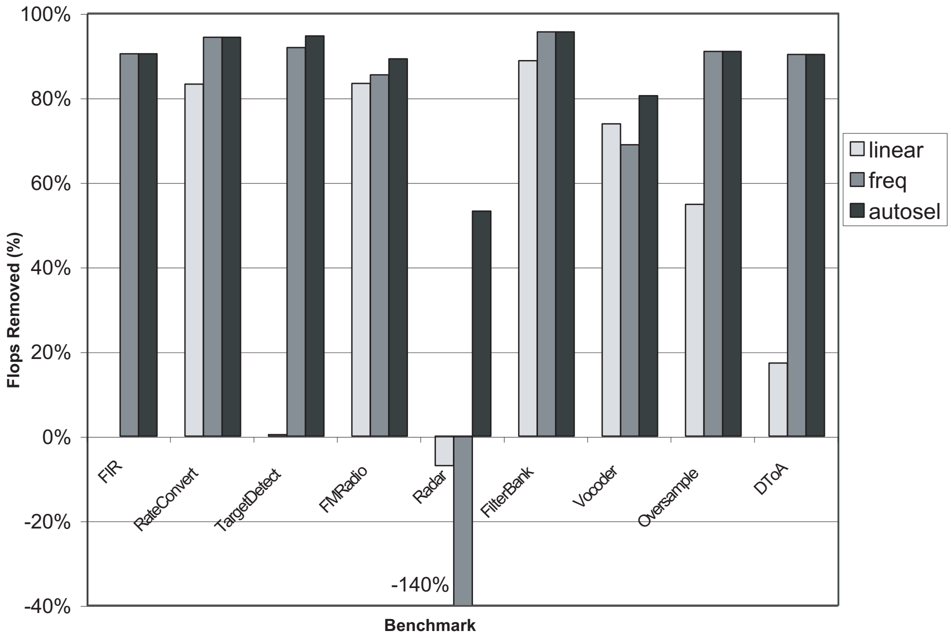


Figure 5-1: Elimination of floating point operations by maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

system and other (FLOPS-preserving) optimizations in the compiler. As shown in Figure 5-1, each optimization leads to a significant reduction of FLOPS. Linear replacement eliminates FLOPS for all except for FIR, TargetDetect, and Radar, while frequency replacement eliminates FLOPS for all except Radar.

The automatic selection option eliminates more FLOPS than either of the other options for TargetDetect, FMRadio, Radar, and Vocoder. The effect is especially pronounced in Radar, where linear and frequency replacement increase the number of FLOPS, but automatic selection decreases FLOPS; the selection algorithm chooses to combine only some of the filters, transforming none to the frequency domain. Automatic selection always performs at least as well as the other two options, which implies that our cost functions have some level of accuracy.

Execution speedup for each benchmark is shown in Figure 5-3. With automatic selection, our benchmarks speed up by an average factor of 450% and by a factor of 800% in the best case. While the graph suggests that frequency replacement almost always performs better than linear replacement, this is not strictly the case; in FM-Radio, Radar, and Vocoder, the automatic selection algorithm obtains its speedup

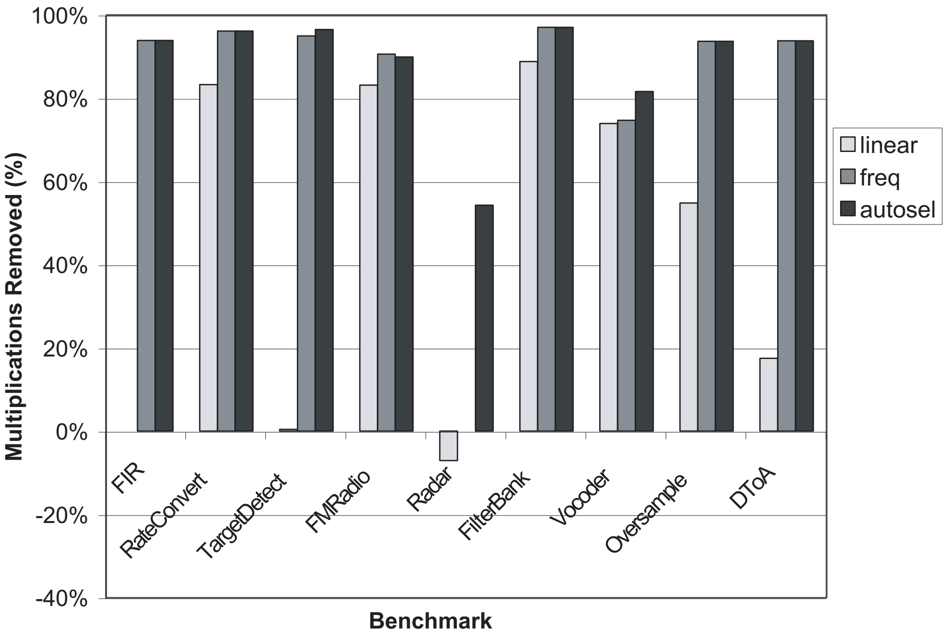


Figure 5-2: Elimination of floating point multiplications by maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

by using linear replacement instead of frequency replacement for part of the stream graph. However, linear replacement does reduce performance for FIR, TargetDetect, and DToA despite reducing the number of FLOPS. We believe that this is due to inefficiencies in our implementation of the matrix multiplication routine, as well as auxiliary effects on the runtime overhead in the StreamIt library. We have experimented with using the machine-tuned ATLAS library for the matrix multiply [36] (see Section 5.4), but performance varies widely: linear replacement with ATLAS performs anywhere from -36% (on FMRadio) to 58% (on Oversampler) better than it does with our own matrix multiply routine, and average performance with ATLAS is 4.3% lower. Note that these numbers reflect our overhead in interfacing with ATLAS rather than the performance of ATLAS itself.

Perhaps the most interesting benchmark is Radar[2] (see Figure B-4 for stream graph). Maximal linear and frequency replacement lead to abysmal performance on

[2]This is the same Radar application presented in [15], with some filters adjusted to work at a coarser level of granularity. This eliminates persistent state in exchange for increased I/O rates. Also, frequency replacement caused an internal error in gcc for this program, so we used egcs 2.91 instead.

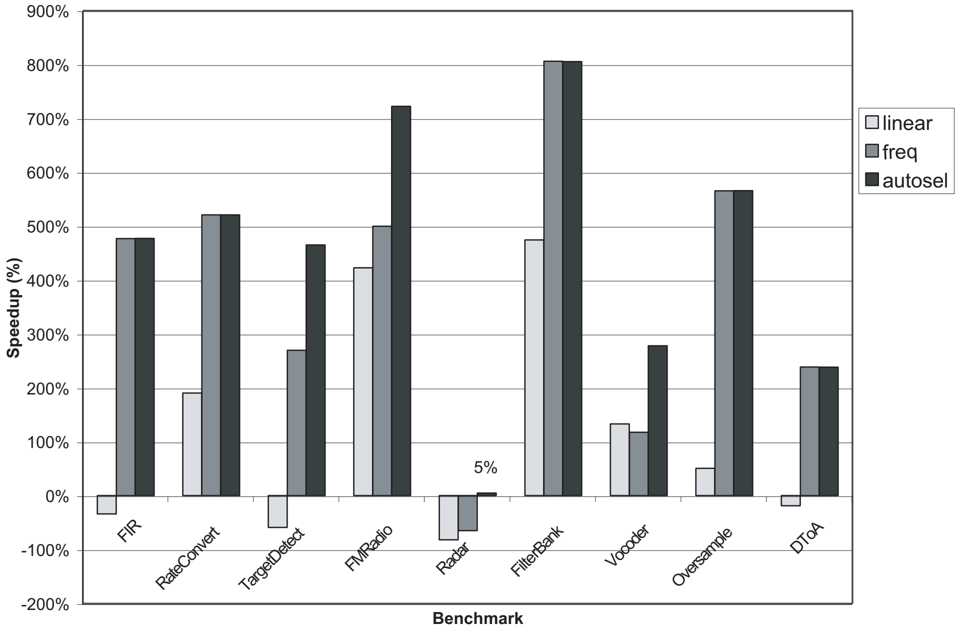


Figure 5-3: Execution speedup for maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

Radar, due to a vector-vector multiply filter named "Beamform" at the top of a pipeline construct. The Beamform filter pushes 2 items, but pops and peeks 24; thus, when the replacement algorithms combine it with a downstream FIR filter, much of its work is duplicated. Moreover, the frequency replacement option suffers from the large pop rates in the application (as high as 128 for some filters), thereby increasing FLOPS and execution time by more than a factor of 30. The automatic selection algorithm is essential in this case: it averts the performance-degrading combinations and benefits from linear combinations elsewhere in the program, resulting in a significant reduction in FLOPS and a 5% performance gain.

5.3 Effect of Combination

While a good portion of our execution speedup and our FLOP reduction come from the efficiency of the FFTW library, the combination methods presented in this thesis inherently reduce FLOPS as well. We ran the same execution time and operation count measurements when combination was disabled. Figure 5-4 compares the per-

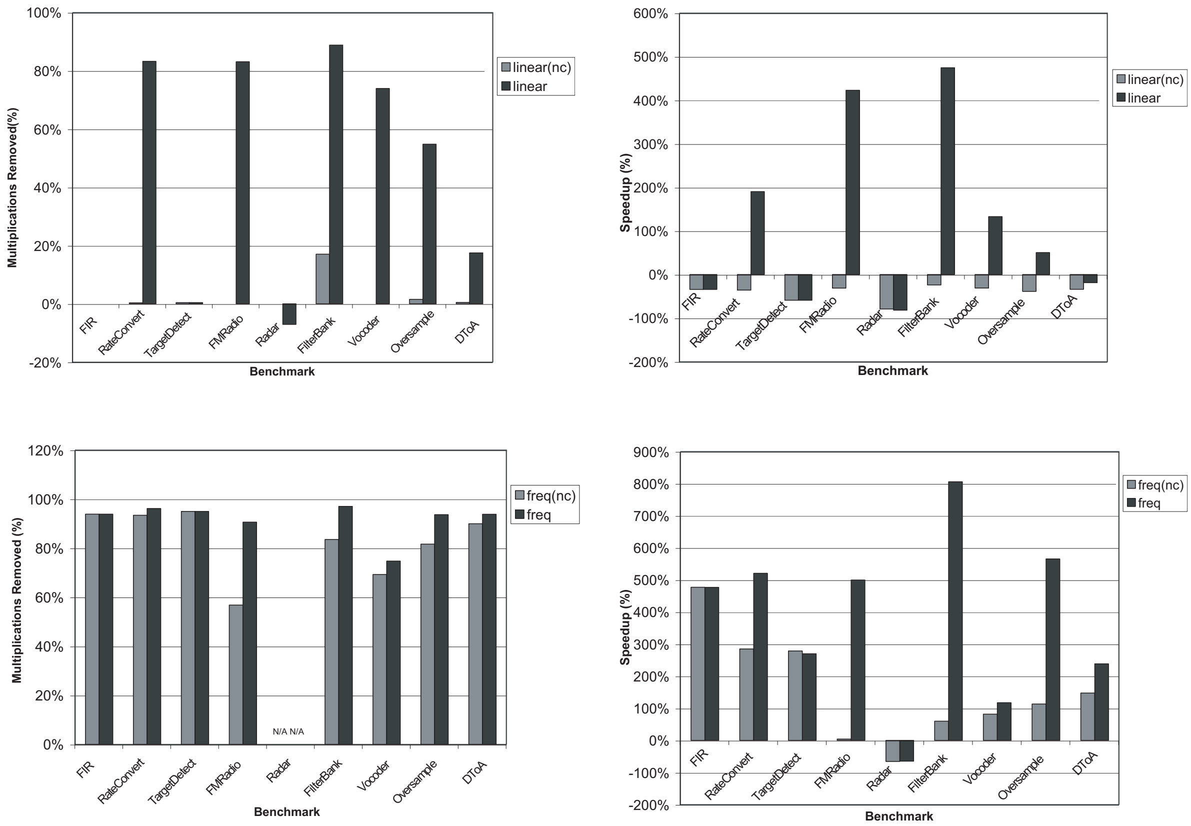


Figure 5-4: Elimination of multiplications (left) and speedup (right) with linear replacement (top) and frequency replacement (bottom) with and without combination. The *(nc)* label denotes that combination was disabled.

formance with no combination to that with combination.

The top two graphs in Figure 5-4 show that for linear replacement, combination leads to most of the multiplication reduction. FilterBank is the only application that has significant multiplication reduction without combination. We attribute the reduction to our implementation of matrix multiply which does not generate unused expressions. Execution time for linear replacement is better in all cases where combination is enabled than when it is disabled except for Radar. We interpret these result as validation of the fundamental strength of our combination algorithms in their own right as well as their usefulness as components of other optimizations (*i.e.* frequency replacement).

The bottom two graphs in Figure 5-4 show the effect of combination on speedup and multiplication reduction when using the frequency replacer. In all of the bench-

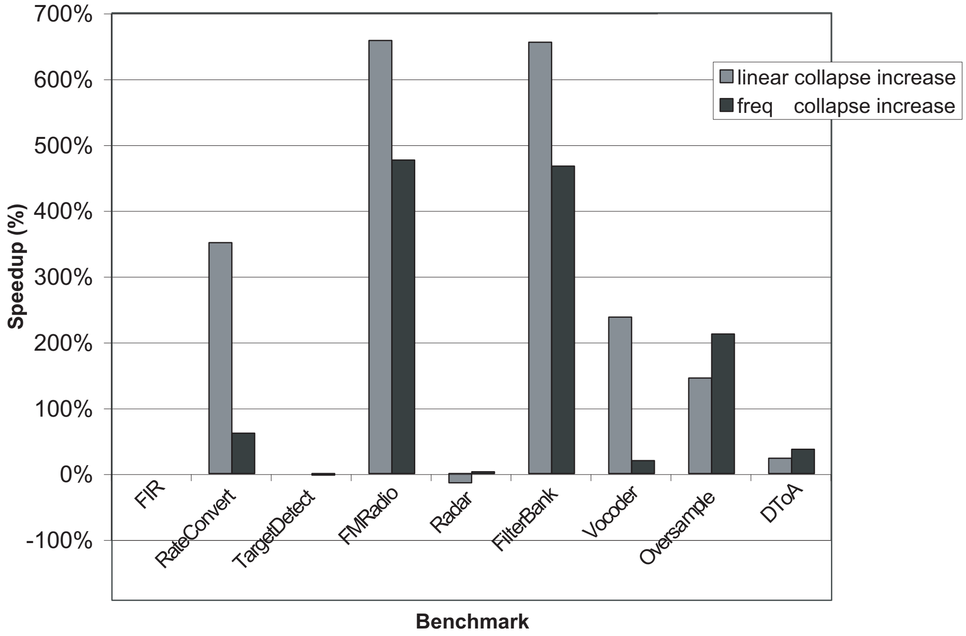


Figure 5-5: Differences in speedup due to the addition of combination with linear replacement and frequency replacement.

---

marks except Radar and FIR, frequency replacement generates impressive reductions even when combination disabled. Enabling combination further improves the reduction for all benchmarks. Execution time is decreased in all benchmarks except TargetDetect. Since the multiplication reduction is the same for TargetDetect both with and without combination, we attribute the 3% decrease in speedup to overhead in the runtime system and variations in our timing platform.

Figure 5-5 summarizes the increase in execution time due to combination for linear and frequency replacement. It is interesting to note that combination makes no difference for the FIR benchmark. This result makes intuitive sense because there are no adjacent linear filters that combination might improve. Due to the nature of the Radar benchmark (described above in Section 5.2), the numbers in Figure 5-5 for Radar reflect the fact that filters with a pop rate other than one were not transformed by frequency replacement[3]. The unit pop rate restriction disallows frequency transformations for the entire Radar benchmark and it explains the lack of multiplication reduction for frequency replacement.

[3]Without the unit pop rate restriction, code size explodes for the Radar benchmark.

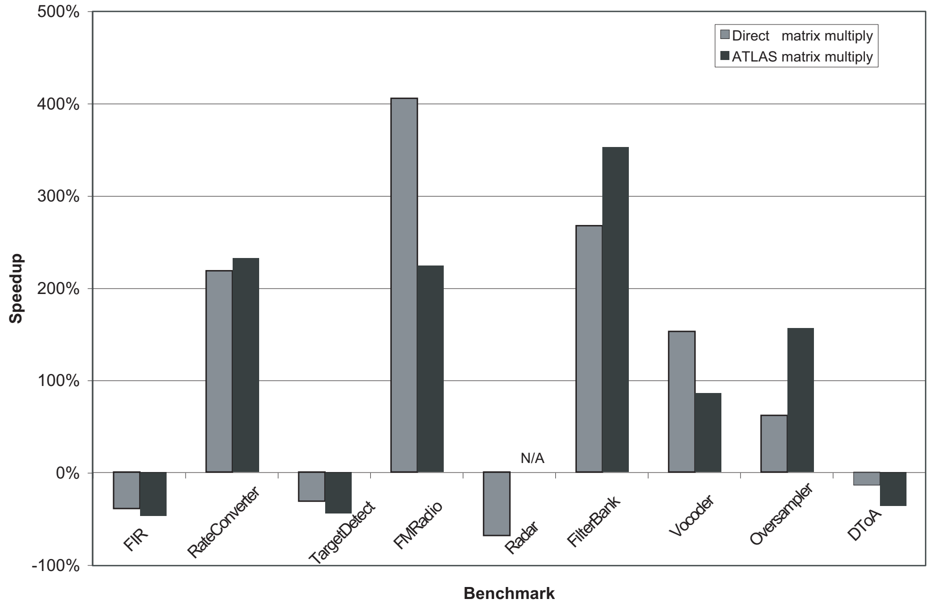


Figure 5-6: Speedups using ATLAS to implement linear replacement.

5.4 Effect of ATLAS

Using the ATLAS linear algebra system to implement matrix multiplies was inspired by the abysmal performance of our linear replacement optimization without combination. The FIR benchmark executes 34% slower with linear replacement even though the code is implementing exactly the same operation as the original program and it requires exactly the same number of FLOPS. This observation implies that the code generation strategy for pure linear forms is sub optimal.

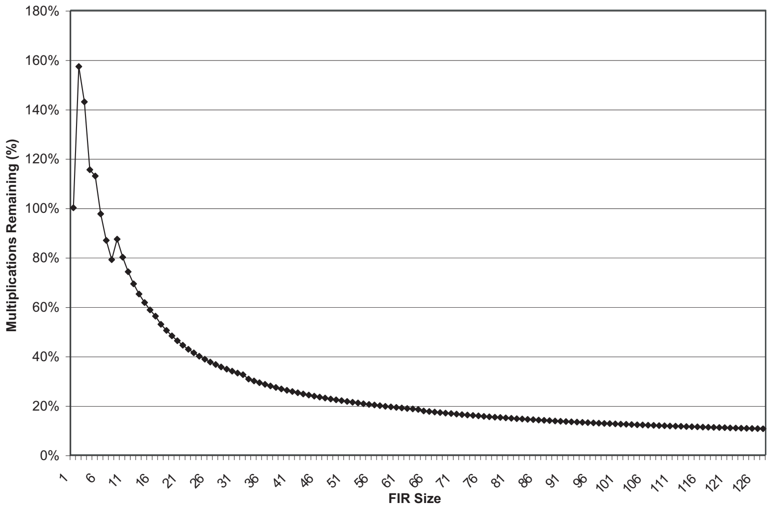
For small filters ($e < 256$), code generated by linear replacement contains one push statement per column of the associated linear node whose argument is an expression of the form $a * \texttt{peek}(i)$ (*i.e.* $\texttt{push}(\texttt{a0} * \texttt{peek}(0) + \texttt{a1} * \texttt{peek}(1) + \ldots)$). For larger filters ($e \geq 256$), we store the matrix coefficients in an array A and compute the result in a loop nest, rather than inlining the computation into a single expression as shown in Figure 5-7. Note that our code is optimized for the case in which a column of A begins and/or ends with a series of zero entries. We avoid storing these zero entries by using a sparse A matrix in which the zeros have been removed from the ends of each column; we also adjust the loop bounds accordingly. This optimization improves performance for a number of our benchmarks. We also experimented with a

```
int sum, count;
for (int j=0; j<numPush; j++) {
  float sum = 0.0;
  // don't multiply the zero-entries
  // at beginning and end of column
  int count = firstNonZero[j];
  int length = lastNonZero[j] - firstNonZero[j];
  for (int i=0; i<length; i++) {
    sum += sparseA[i][j] * peek(count);
    count++;
  }
  sum += b[j];
  push (sum);
}
```

Figure 5-7: Code for diagonal matrix multiply for large filter sizes.

sparse matrix multiply in which all of the zero entries are removed and an additional array is used to index the non-zero elements. However, the overhead incurred by this level of indirection outweighs the benefits over the programs in our benchmark suite. In the uniprocessor backend, a peek expression is transformed into a complicated array access whose address calculation includes a logical bitmask operation. As the original program contains the same number of peek expressions, complicated processor instruction scheduling interactions are probably to blame for the poor performance.

The execution slowdown doesn't represent any inherent penalty due to our optimization technique; It merely represents a deficiency in the specific code we generate. We experimented using the ATLAS machine tuned linear algebra library to do the necessary computation instead of generating C code from within the compiler. Figure 5-6 shows execution speedups when using the linear replacement optimization with ATLAS compared to the original code generation strategy. In some applications ATLAS helps (RateConvert, FilterBank and Oversampler) but in the others it has no effect and even decreases performance (FIR, TargetDetect, FMRadio, Vocoder, and DToA). Again, this is not due to any deficiency in ATLAS but rather in the overhead incurred in our interface with it.



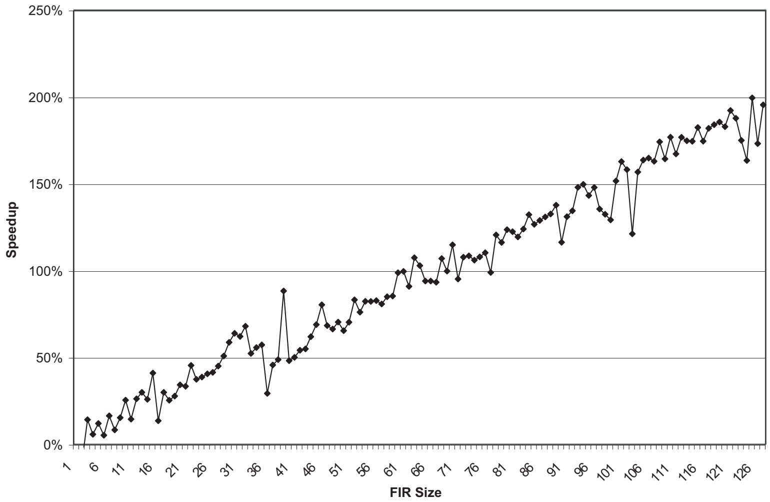


Figure 5-8: Elimination of floating point multiplications (top) and speedup (bottom) with frequency replacement as a function of problem size for the FIR benchmark.

5.5 FIR Scaling

Because the performance improvement due to our frequency replacement optimization is a function of the problem size, N, the extent to which problem size matters for our benchmarks is of interest. To determine the effect of problem size on performance we varied the size of the FIR benchmark from 1 filter coefficient to 128 filter coefficients and measured the multiplication reduction and the execution speedup for each size. Figure 5-8 shows the results obtained from this experiment. The original FIR benchmark contains 256 coefficients, but we do not show the plot to 256 co-

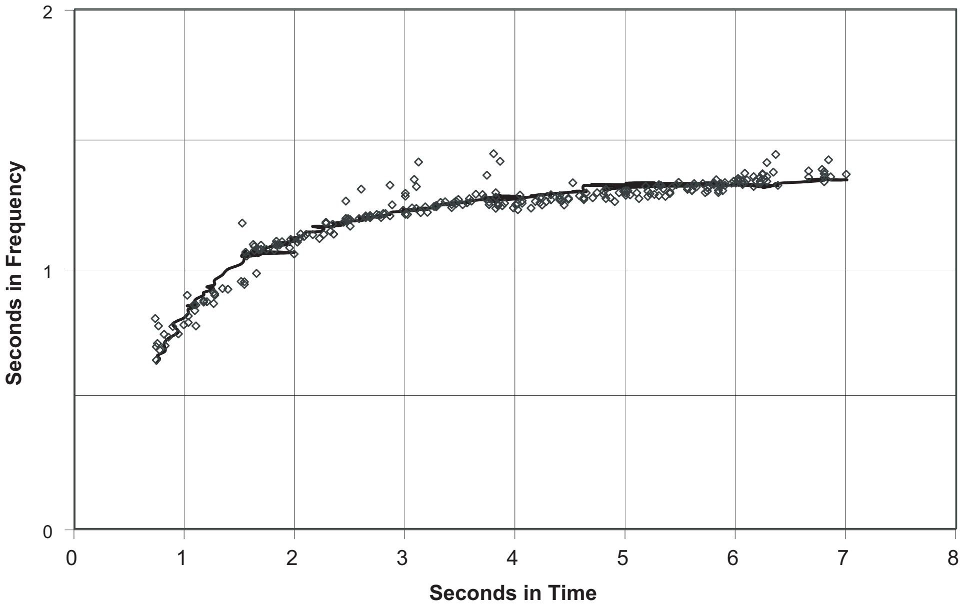


Figure 5-9: Scatter plot of original execution time versus post optimization execution time for FIR scaling experiments. Plotted as a solid line is the cost function used with the automatic selection algorithm.

---

efficients because all of the pertinent features appear closer to the origin. The top graph in Figure 5-8 shows the multiplications remaining after the optimization was applied and demonstrates good agreement with the theoretical prediction[4] of $\frac{\lg(N)}{N}$. The bottom graph of Figure 5-8 shows execution speedup. The graph is more noisy than the multiplication reduction graph and it looks approximately linear. However, in Figure 5-9 we plot the timing data as a function of original execution time and post optimization execution time. It is much easier to see the expected $\lg(N)$ shape[5]. Figure 5-9 also plots the cost function used by our automated selection algorithm which is shown in the following equation.

$$\mathit{freqVal}(\mathit{firSize}, \mathit{timeVal}) = 0.65 + \ln\left(1 + \frac{\mathit{timeVal} - \mathit{timeVal}_0}{1 + \frac{1}{50}\mathit{firSize}}\right)$$

[4]The theoretical $\frac{\lg(N)}{N}$ prediction is due to the $N\lg(N)$ frequency implementation compared to the N^2 original implementation.

[5]We expect $\frac{N}{\lg(N)}$ speedup if speedup is inversely proportional to multiplication reduction.

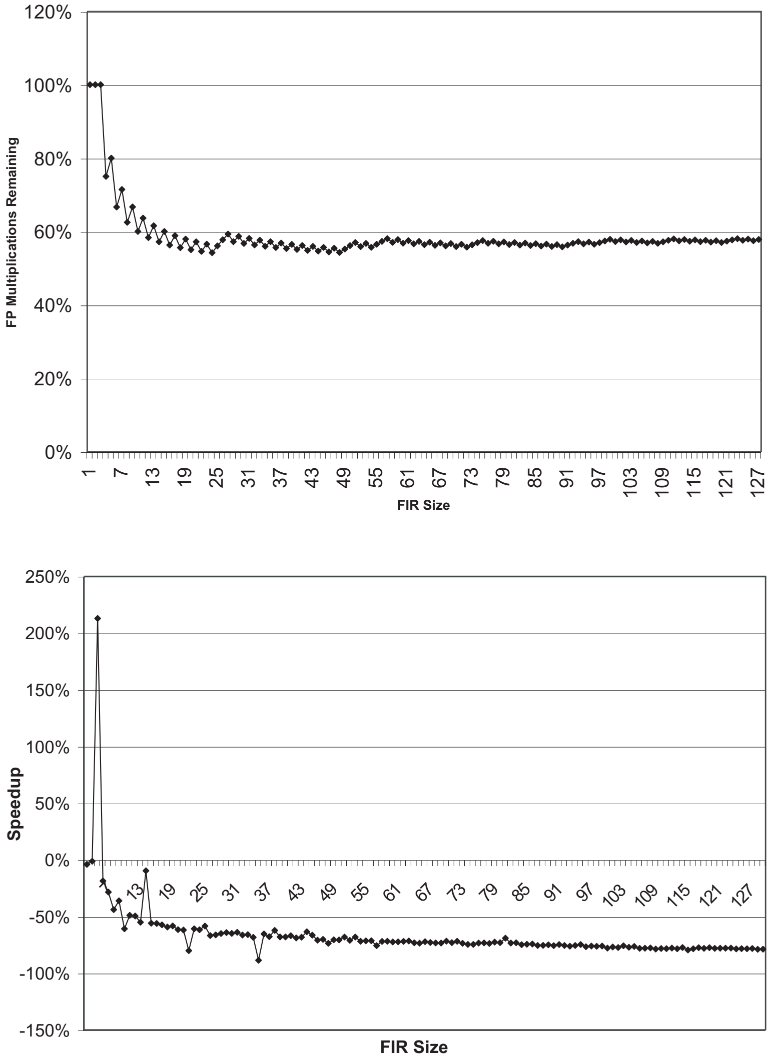


Figure 5-10: Multiplications remaining (top) and speedup (bottom) after redundancy replacement as a function of problem size for the FIR benchmark.

5.6 Redundancy Elimination

We now present simple measurements for the redundancy removal optimization described in Section 4.2. Because of the extensive state required for our implementation of redundancy removal, the code size for the benchmark programs grows out of control when we apply the optimization. To understand the effect of redundancy removal on performance, we measured multiplications and speedup as a function of FIR size. The FIR benchmark has approximately 50% redundancy due to the fact that A is a symmetric column vector yet the program was small enough that `gcc` compiled it without complaint.

The results of the experiments are shown in Figure 5-10. Redundancy removal does a good job of removing multiplications as shown in the top graph. However, the overhead necessary to affect this removal erases any potential benefit and slows the program down. Even worse, as the program size grows and the amount of removed redundancy increases, the program slows down even more. This decrease in performance as a function of FIR size is due to the fact that as the redundancy increases, more time must be spent to load and store cached values. Since the overhead to load and store is greater than the savings from removing the multiplication the overall program becomes slower.

One interesting feature of the top graph in Figure 5-10 is its peculiar "zig-zag" shape. The saw-toothed curve is not an accident. It results from the difference in redundancy of even and odd sized FIR filters. In an odd sized filter the values computed with the center coefficient can't be reused, while in an even sized filter there is no middle coefficient and all values can be reused. Therefore, if N is odd, after Transformation 7 is applied a filter of size $N + 1$ will have fewer multiplications than a filter of size N even though the problem size is larger.

5.7 Radar Scaling

As mentioned previously, the Radar benchmark is challenging for our optimization techniques. Even more interesting is that our optimizations' performance depends on the problem configuration. The Radar benchmark as written has 12 "Channels" and 4 "Beams" representing 12 data sources being focused in 4 different directions. Referring to Figure B-4, "Channels" are in the top of the application and "Beams" are in the bottom. Figure 5-11 shows how maximal linear replacement changes the number of multiplications as a function of problem size. As the parameters grow, linear replacement does worse. The degradation due to increasing the number of Beams is much more pronounced than is increasing the number of Channels.

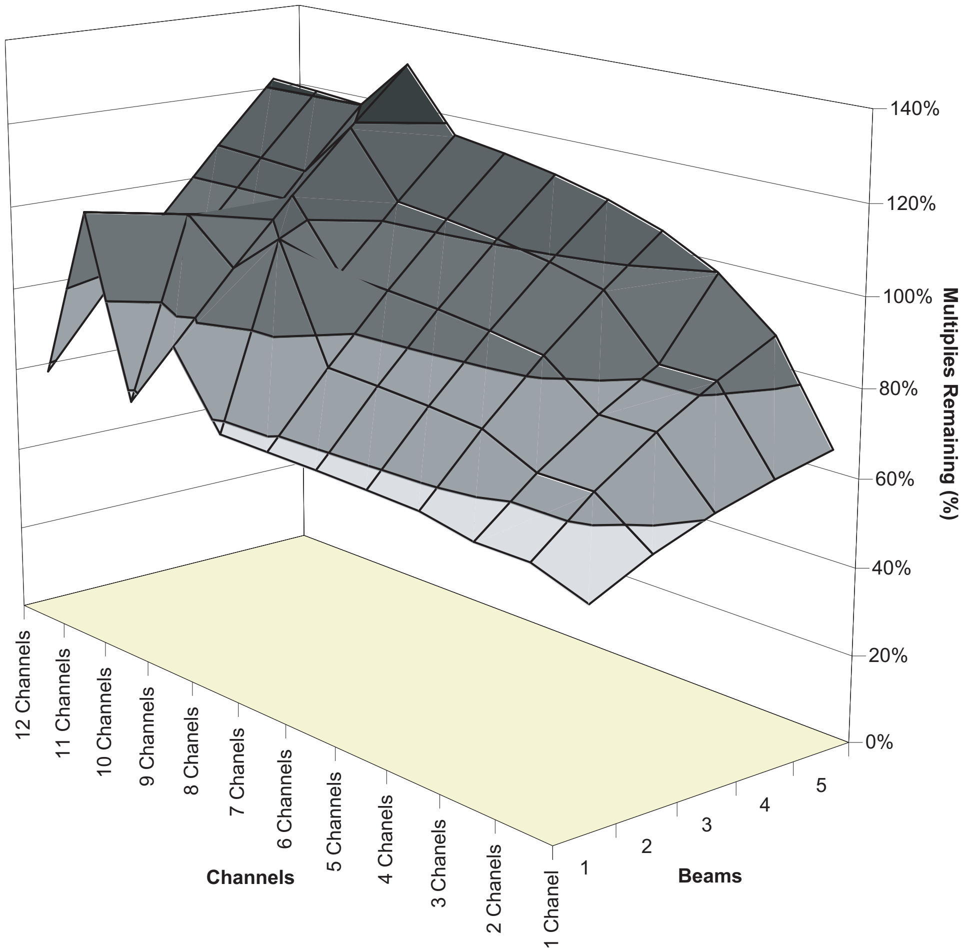


Figure 5-11: Multiplication reduction with maximal linear replacement as a function of problem size for the Radar benchmark.

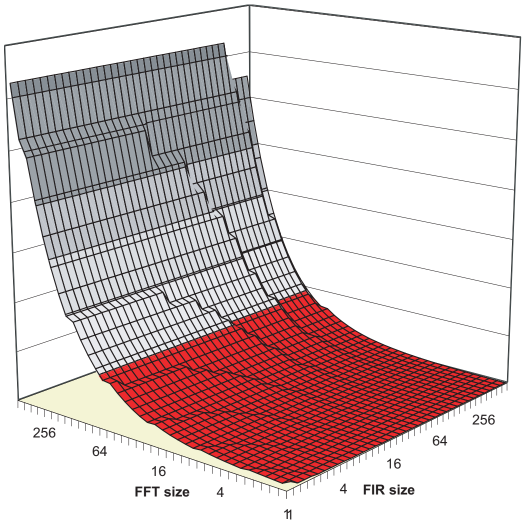
5.8 FFT Savings: Theory vs. Practice

The strength of our results rely only partially on the efficiency of FFTW. Using the optimized frequency transformation (Transformation 6) over the naïve transformation (Transformation 5) also improves performance. To quantitatively evaluate the performance improvement due to various improvements in our frequency replacement technique we present the experiments summarized in Figure 5-12.

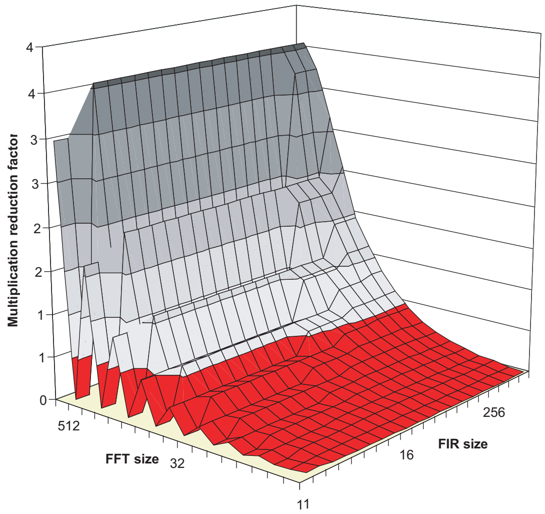
Figure 5-12 shows the effects of different frequency transformation strategies on post transformation multiplications. To highlight the differences between methods, we report the *multiplication reduction factor* which is defined as the number of multiplications originally required per output over the number of multiplications required per output after applying the specified frequency transformation. The x axis is the size of the FIR and the y axis is the size of the FFT. The FFT size was manually specified instead of automatically chosen.

- Figure 5-12 a) shows the theoretical reduction.
- Figure 5-12 b) shows the multiplication reduction using the naïve transformation (Transformation 5) and a simple FFT implementation.
- Figure 5-12 c) shows the improvement from Figure 5-12 b) due to the optimized frequency transformation (Transformation 6).
- Figure 5-12 d) shows the improvement from Figure 5-12 c) due to using FFTW to perform the FFT.

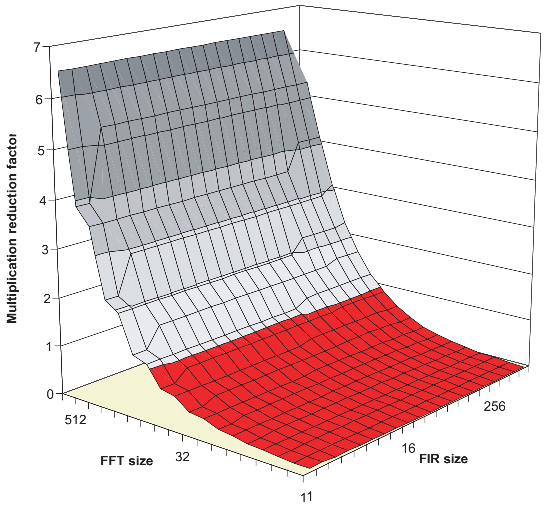
From Figure 5-12 we can see that the optimized transformation improves performance by a factor of 1.5 over the naïve transformation and using FFTW improves it by a factor of 6. While FFTW provides a large performance boost, our optimizations are also worthwhile in their own right.



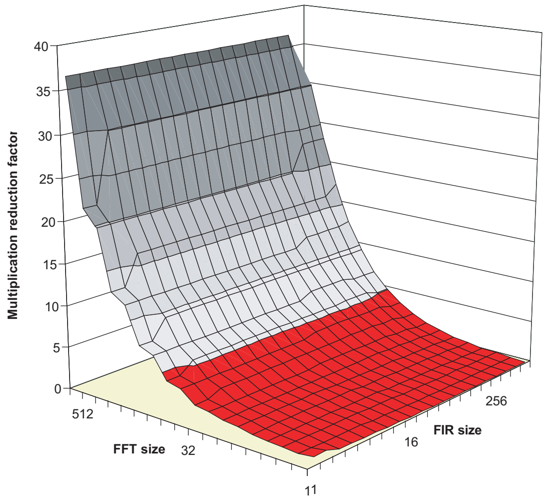
a) Theoretical



b) Naïve



c) Optimized



d) Optimized (FFTW)

Figure 5-12: Elimination of floating point multiplications with frequency replacement as a function of problem size and FFT length for the FIR program using different transformation strategies.