List of Figures

List of Tables

Contents

[4] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *J. Acoustic. Soc. Am.*, 87(4):1738-1752, April 1990

[5] H. Hermansky and N. Morgan. RASTA processing of speech. *IEEE Transactions on Speech and Audio Processing*, 2(4):578-589, October 1994

[6] N. Morgan. The Ring Array Processor (RAP): A multiprocessing peripheral for connectionist applications. *Journal of Parallel and Distributed Computing*, 14:248-259, 1992.

[7] H. Bourlard and N. Morgan. Merging multilayer perceptrons & Hidden Markov Models: Some experiments in continuous speech recognition. In E. Gelenbe, editor, *Artificial Neural Networks: Advances and Applications*. North Holland Press, 1991.

[8] H. Bourlard and N. Morgan. *Connectionist Speech Recognition - A Hybrid Approach*. Kluwer Academic Press, 1994.

[9] H. Hermansky, N. Morgan, A. Baya, and P. Kohn. RASTA-PLP speech analysis. *ICSI TR-91-069*, December 1991

[10] K. Asanovic and N. Morgan. Experimental determination of precision requirements for back-propagation training of artificial neural networks. *ICSI TR-91-036*, October 1991

[11] D. Anguita and B. A. Gomes. MBP on T0: mixing floating- and fixed- point formats in BP learning. *ICSI TR-94-038*, August 1994

5 Future Work

Because so far only a part of the RASTA processing has been implemented on SPERT, a natural next step is to conclude the implementation and carry out a global comparison, from the point of view of functionalities and performance. In particular, it would be worthwhile to assess the impact of a fixed point feature extractor on the recognition stage of the system. In other words, as long as there is a nonzero error between fixed and floating point versions, we can not exclude a significance of the error on the following stages of recognition.

6 Conclusions

This work is a pilot study on the problem of efficiently vectorizing speech processing algorithms. We have considered the main computational characteristics of the RASTA-PLP processing and their possible mapping over a vector fixed point microprocessor.
We found that a limited precision vectorized processing performs almost as well as the standard floating point processing, whereas a significant improvement in time performance is obtained.
This work merely scratches the surface of the many issues in this area.

7 Acknowledgements

Special thanks to David Johnson for providing all the software environments and the simulators, for assistance with them, for all the help and support and valuable comments, and for making his experience and ideas readily available throughout.
Also, thanks to Warner Warren for the FFT and power spectrum routines and for useful contributions on speech processing, filter banks and precision, Eric Fosler for providing the WSJ0 database, John Hauser for the log and exp functions, and other contributions to the Torrent libraries, Krste Asanovic and Jim Beck for several interesting discussions on *T0* architecture, and Nelson Morgan for suggestions on the whole problem.

References

[1] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. SPERT-II: a vector microprocessor system and its application to large problems in backpropagation training. *Submitted to NIPS 95*

[2] K. Asanovic and D. Johnson. *Torrent Architecture Manual*

[3] K. Asanovic and J. beck. *T0 Engineering data.*

on precision.

4.5 Integration

The steps illustrated above have been integrated and a complete implementation has been carried out on the simulator.

The processing is done frame by frame, as in the usual speech processing. The floating point input data are converted into fixed point 16 bit numbers using an exponent value of 16 (because the input data are 16 bit samples). The Hamming windowing has not been implemented for this experiment. We used the FFT routines with 160 points. This choice has the advantage of not requiring zero padding of input data, usual procedure with radix-2 FFT. After that, the SPERT version of power spectrum is also used. After the power spectrum we have a 32 bit representation. Then we used the 32 bit fixed point dot product routine to carry out the critical band analysis. We considered an input exponent of `42` and an output exponent of `46`. This is because the vector length of the dot products to be executed ranges from 7 to 57, according to the parameters used. Then the optimal choice (see Table 7) is to have:

$$expres - (expinp + expcoeff) = 3$$

The critical band filters coefficients are in the range (0.0 - 1.0]. Again, the rule is to choose the exponent as low as possible. The choice $expcoeff = 0$ is not possible because the extremes of the range are not representable, so we chose $expcoeff = 1$. This led to the choice of input and resulting exponents as stated before.

The dot product routine is called every time for every filter to be processed. Then the data are converted again into a floating point representation.

Note that a SPERT board has been installed into the SPARC5 (icsib29), with 0.5 Mbyte of memory and clock rate at 40 MHz. Therefore the same implementation ran successfully on the true SPERT board, other than on the simulators.

Note also that so far we have not implemented several parts of the complete RASTA processing, most notably the temporal filtering and the cepstrum analysis. These are left as future work.

	dft32	dft64	dft32x8	dft32x16	Tdft32x32
ideal	256	512	2048	4096	8192
actual	330	792	2742	5800	11720
%utilization	78	65	74	71	70

Table 8: Execution time (in cycles) and utilization for some DFT32xN routines

	dft8x32	dft16x32	Tdft32x32
ideal	1024	4096	8192
actual	1307	5816	11720
%utilization	78	70	70

Table 9: Execution time (in cycles) and utilization for some DFTNx32 routines

The following tables list the execution times and processor utilization of the FFT routines. Table 8 lists the 32 point DFT32xN routines, whereas Table 9 lists the N point DFTNx32 routines.

The DFT32xN routines are instruction issue bound. Each routine has an inner loop which executes 4 vector load multiply accumulate operations. Thus, each pass through the inner loop requires minimum of 12 instruction cycles and and 16 execution cycles. This means that a perfectly balanced inner loop would have no more than 16 instructions. Unfortunately, the required scaler memory operations and loop control operations push the instruction count up to about 19, resulting in a maximum utilization of 16/19 or 84%.

The DFTNx32 routines listed in the table are memory bandwidth bound. At present, there is no way to control the alignment of input data, so 5 cycles are required to read each vector from memory. This pushes the inner loop length up from 16 to 20, so the highest possible efficiency is 16/20 or %80. Again, this figure is further reduced by bookkeeping operations.

A power spectrum routine has also been implemented, squaring the real part and the imaginary part and then summing up.

Concerning the numerical issue, all the computation is done with the exponent as low as possible. If saturation occurs, the computation is executed again with a higher exponent. This might result in a slow-down of the computation, and it also implies that obtainable performance could depend

The resulting matrix contains the DFT of the transpose of the input matrix. Finally, there is an optional data shuffling stage which leaves the data in ascending order. The reordering step is often not necessary. For instance, if one is using the DFT as intermediate in the computation of a convolution or a correlation, then the reordering is skipped since one can always find an inverse FFT compatible with ordering of the output data. Note that these routines are specifically designed to ensure that most computations are carried out on length 32 vectors. Generally, SPERT routines run most efficiently on vectors of maximum length, which is exactly 32.
The entire process is illustrated in Figure 5.

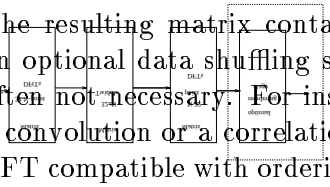


Figure 5: FFT algorithm

4.4.1 Execution time and Processor utilization

The minimum execution time for computing the FFT using the technique here described is (in cycles):

$$256N + 16N^2 + 24N$$

The first factor refers to the 32 point DFTs [15], the second term refers to 32 N-point DFTs, and the third term refers to twiddle factors. The first two terms, 256N + $16N^2$, represent the cost of their respective operations computed at 100% processor utilization, omitting cycles used for loop maintenance, subroutine overhead etc. The twiddle computation can only use one pipeline at time, so the 24N figure represents the minimum cost at 50% utilization. We define the processor utilization in the usual way, that is the ratio of the number of cycles in which the arithmetic pipelines are used to the total number of available cycles in the routine.

[15] In the case of the DFT32xN the calculation requires 32 vector multiply accumulate operations, and each length 32 vector multiply accumulate requires 4 cycles when fully pipelined (i.e. 4 add cycles, 4 multiply cycles and 4 memory cycles running concurrently). This done for 32 vectors yielding 128 cycles/row. Therefore, a complex-complex transform requires at least 256 cycles/row, so we obtain a minimum of 256N cycles

fact having zeroes can cause discontinuities and exceptions in the following stages.

- Usually after an FFT, the DC components carry rather high values, often exceeding all the other values by a considerable amount. As such components are irrelevant to the purposes of the speech recognition systems, it could be useful to leave out such components. This can reduce the range of the input of critical band analysis. For this and other reasons, an optional highpass (45 Hz) filter has been introduced at the beginning of the RASTA processing.
- It is always safe to choose the exponent proportional to the log of the vector length.

After the analysis of all these figures, we decided to implement the FFT using 16bit representation. The following step, namely the power spectrum, is essentially a sum of squares. Therefore it forces the conversion to a 32 bit representation. In fact the critical band analysis was carried out in a 32b representation. The following step is the logarithm, which causes a considerable decrease in the range. Therefore after the logarithm we have again a 16b representation.

4.4 FFT

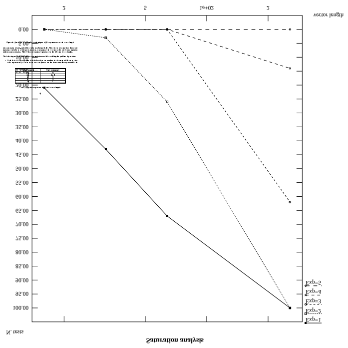
As shown in Section 4.1, Fast Fourier Transforms represent one of the most compute-intensive section of the whole processing, so here was where effort was mostly concentrated.

In the following[13] we describe a number of mixed radix FFT routines designed to run on the SPERT architecture. These routines are designed keeping in mind the two main objectives, which are to maximize computational efficiency, and maintain high numerical precision using fixed point arithmetic. To this purpose, the following routines have been implemented: FFT32, FFT64, FFT96, FFT160[14], FFT256, FFT512, FFT1024. The names of the routine refers to the number of points on which the FFT is computed.
All these routines were implemented as full complex-complex transformations, so that they should also be useful for correlations and convolutions. Moreover, very little additional computation is required to convert an N point complex-complex transform into a 2-N point real-complex transform. Note that the mixed radix implementation can be a considerable advantage, because in many cases the usual zero-padding of the input data could be no longer necessary.

All the routines follow the same basic procedure. First the data are split into length 32 vectors, effectively organizing the data into a 32xN matrix. Then, an N point DFT is computed on each row of the data matrix. The resulting 32 N-point DFT's are then multiplied by the appropriate twiddle factors. Next, N 32-point DFTs are computed on the columns of the resulting matrix.

[13] The FFT part has been studied, designed and developed by Warner Warren. It is reported here for sake of completeness

[14] FFT128 is not present because the technique used to compute the FFT does not allow to compute the required 4 point DFT in an efficient manner


Saturation analysis
% zeros
vector length
Exp=1
Exp=2
Exp=3
Exp=4
Exp=5
100.00
95.00
90.00
85.00
80.00
75.00
70.00
65.00
60.00
55.00
50.00
45.00
40.00
35.00
30.00
25.00
0.00
2
5
1e+05
2

Proportion of the Real Numbers Maximum (Averaged)
Total Quadratic Error
Average Quality

RANGES			
	Lowest negative value	Highest positive value	Lowest absolute value
Input data[11]	-	10^4	1
Hamming window	-	10^4	10^{-2}
Power spectrum	-	10^{12}	10^{-8}
Critical band	-	10^{12}	10^3
Logarithm[12]	-	10^3	10^3
Rasta temporal filter	-10^1	10^1	10^{-5}
Exponential	-	10^5	10^{-7}
Equal loudness + cube root	-	10^1	10^{-4}
Cepstrum	-10^1	10^1	10^{-5}

Table 6: Order of magnitude of the output data ranges of stages of RASTA processing

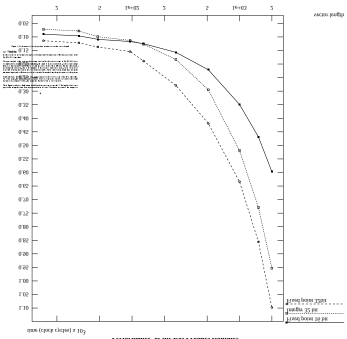
considerably wide. Take for example the output of the power spectrum. In this particular case the range is 20 orders of magnitude wide, so apparently as much as 68 bit would be necessary to properly represent these data. Naturally this is not possible with the SPERT architecture.

The following series of experiments investigate the effect of changing the exponent of the fixed point input data and coefficients. The difference in accuracy between the fixed point routine and an equivalent floating point routine is also investigated. The exponent of the MSB (sign bit) was varied in the range 0-8 for a precision of 16 bits. For example, an MSB exponent of 1 would allow data in the range ± 2. As what is relevant to the assembler routine is the difference:

$$resexp - (inp1exp + inp2exp)$$

it was simpler to choose the two input exponents as zero and then to tune up by varying only the output exponent. So random numbers in the range -1 - +1 have been used as input and as coefficients. The same tests have been carried out on SPERT and on SPARC. Figure 2 and Figure 3 show the difference between the floating point result and the fixed point result, varying the MSB exponent for several vector lengths. In the same condition we executed 100 tests. The two figures report respectively the average difference and the maximum difference found for every particular exponent.
Additionally, Figure 4 shows the number of tests with saturation flag set, during the same experiments described above.

As expected, with a higher exponent the accuracy is lower, but no saturation occurs. Conversely, with lower exponent the accuracy might be better, but more often saturation occurs. This partic-



Performance of the Dot Product Routines

Note that, if the saturation register is updated, a saturation flag is set. Such flag means fatal intermediate saturation, and the result can be completely wrong (this is indicated by a "t" at the end of the name of the routine). In other words, the exponents should be properly set in order to avoid this case of not recoverable saturation.

In the fixed point case, the user must supply exponents for the operands and an exponent for the result. These exponents are the exponents of the MSB (sign bit) of the number. Actually what is relevant to the routine is only the difference between the result exponent and the operands exponents, because a multiplication is a sum in the exponents domain. So this difference contains enough information to calculate the appropriate right shift to be taken.

To implement these dot product fixed point routines, other than all the observations already made with regard to the integer routines, two more options have been considered:

- multiply, shift and accumulate. It is fast: all can be done in one operation because the fixed point pipeline allows multiple operations. It might cause a considerable loss of precision, because the shift is executed every time before the sum, whereas the sum can cause high variations to compensate, so no saturation would happen on the final result. Anyway, this is the way chosen for matrix vector multipliers.
- multiply and accumulate over 64 bit, and then shift. More precision is obtained, but this method is surely slower.

So far the implementation has been carried out only for the first choice. A natural next step would be the implementation of the second choice too, in order to make a performance/precision comparison.

These routines have been tested extensively using the random libraries to generate input data. The results have been compared with analogous C routines. However, because of the saturation flag issue, in some cases the results can be different. In fact, the saturation flags calculated in C and assembler can be raised in different situations because of the different ways to calculate them. In C the sum is executed element by element, whereas in assembler 32 elements are summed simultaneously, so that compensation can occur. It turns out that there are cases in which the flag is raised for the C case, and not raised for the assembler case. Moreover, in the cases where are both set, the results are not comparable, because both results are not correct.

Figure 1 indicates the performance of the routines for several vector lengths.

The run-time values in clock cycles have been obtained using the RTL simulator for T0.

- The most critical resource is memory, so try to keep the memory pipeline always busy.
- Pipeline the loops: that means to execute the first loads or other operations outside the loop, then start the loop and finally execute the last operations outside the loop itself.
- Insert scalar instructions in between vector instructions. When the vector unit is busy, in this way we manage to exploit the scalar arithmetic unit.
- It is always useful to put the short case (vector length less than 32) first for better cache performance.

The global strategy followed to implement the dot product, in all the four cases described before, has been the following (considering vectors of any length):

1. multiply the first 32 elements of the two vectors outside the loop,
2. multiply another slice of the vectors.
3. sum with the previous product and accumulate. Again step 2 until the end of the input vectors is reached.
4. At this point the sum of all the elements of a vector length 32 must be found. It is possible to do that with five vector add operations, by splitting the original 32 elements vector in two halves of 16 elements, summing them up, and iterating on vectors of half the length until a scalar is obtained.

Fixed point operations require a further analysis. The add and multiply fixed point instructions are primarily used to implement scaled, rounded and clipped fixed point arithmetic. The necessary information is supplied by a scalar register termed the "configuration register". A saturation status register `vsat` is updated by fixed point arithmetic operations. Note that, according to the values of the configurations register, within the same operation, say add, it is possible to execute logic operations, shift left and right, clipping, sign extension, rounding. Note also that, even for the case of the integer routine, having to deal with sign problems, it is necessary to use fixed point operations.
For the particular case of integer routines, the following choices have been made:

- for the multiply, the low half word is effectively unsigned, that means zero extended; the high half word is signed, that means sign extended. No shifting or clipping is needed. The round bit is added.
- for the add: shift left one of the arguments of 16 positions, then add. No rounding or clipping is needed.

4.2.1 Dot Product Routines

To this purpose four dot product routines have been developed. The routines are different by the kind of input they can take: integer 16 bit, integer 32 bit, fixed point 16 bit, fixed point 32 bit. They can take as input 2 vectors of any but equal length. Every element of the vector is represented by the number of bit indicated in the routine name. The output is a scalar also represented in the same way. The names of the routines follow the usual naming convention of the whole Torrent `fxlib` library.

The following describes some of the issues encountered in the design and implementation phases of these routines. Besides the specific case, this should be of interest for others *T0* programmers, having to deal with other algorithms to vectorize.

As usual, it has been necessary to separate the case of short vectors, having less than 32 elements, from the case of long vectors. This is because the registers of the vector unit hold only 32 element each 32b wide. Therefore, for longer vectors, it is necessary to work at slices of 32 elements at a time.

The T0 CPU is a MIPS-II compatible 32b integer datapath. However, the multiplier within the vector unit (VP0) can perform up to 8 16bX16b multiplies per cycle, so it can handle only 16b data. Therefore in order to multiply 32b vectors it is necessary to split the multiplication in 4 steps, separating every data in the low and high halfword.

The code has been sheduled by hand in order to obtain optimal performance, paying attention to interlocks, delay slots and structural hazards. In fact, even if all the CPU instructions take a single cycle to issue, in many cases the results of the instructions are not available to the instruction issued in the following cycle. This is valid for example for branches, loads, multiplies and divides. In these cases, there is a minimum number of instructions that must be scheduled between dependent instructions to avoid an interlock.

Another very common situation is represented by the vector unit hazards. T0 has two vector pipelines, VP0 and VP1. All multiply instructions must execute in VP0, all other instructions can be executed in either pipeline. Each of the two vector arithmetic unit completes 8 element operations per cycle: that means that is busy for 4 clock cycle when processing an arithmetic operation of a vector of length 32. The CPU will stall if it tries to access a vector unit when its pipeline is busy. The memory unit reads and writes 16 bytes per cycle into 8 ports of the vector register file: when transferring contiguous vectors that means that it takes 8 cycles to move 32 words. The CPU will stall if it tries to access the memory when the memory pipeline is busy.

Here are the general and heuristic criteria we used to schedule the code for this particular application:

These figures show that a gain of more than 30% is possible in case of optimal and efficient implementation of the library functions log, exp, sin, cos, pow on SPERT. A further improvement should be considered due to the library calls present in the initialization part and not removed in this experiment.

4.2 Critical band analysis

Currently the implementation of the critical band analysis involves the integration of the power spectrum in sections. In particular, the spectrum is first warped along its frequency axis into the Bark frequency Ω. The resulting warped power spectrum is then convolved with the power spectrum of the simulated critical band masking curve.

In practice, this turns out to be a body of dot products, of different lengths, and different starting points in the input data. Considering the input spectrum as a vector, and the output data as another vector, several choices are possible about how to carry out the implementation of this part on SPERT. Note that the typical input frame length is 129 samples, and 18 samples as output, so these are not the long vectors most useful to a vector machine.

In details these are the possible choices:

1. To implement it as many dot products. This requires the design and development of a bunch of dot product routines, not yet present in the library. In fact it would be necessary to have a routine for both the cases integer and fixed point, and for the 16b and 32b input. A further better choice would be the implementation of an indexed dot product, capable of picking up the data in a vector at selected indexes. Another good idea is to optimize these dot product routines for short vectors.

2. To implement it as a matrix vector multiplication. The routine for the matrix-vector multiply is already present in the library, but it is optimized for long vectors. This choice implies the building up of a sparse matrix collecting all the coefficients required for the simulated critical band masking curve. A further step would be to optimize the matrix-vector multiply routine for shorter vectors.

Both the ways 1 and 2 present their own advantages and disadvantages. So far, only the first approach has been followed, and the results illustrated in the following subsections. But it would be interesting to carry out the implementation according to the second approach, and compare the obtainable performance.

Another critical issue to be considered in dealing with this processing is the impact of reduced input and coefficients precision. The effect of having fixed-point data, instead of floating point has been investigated, and in particular the effect of changing the exponent and the precision of both input and coefficients.

RUN-TIME DIFFERENCE	
Library functions	Dummy functions
28.0	19.1

Table 5: Run times for the sparc20 experiment, with and without library functions (in seconds)

starting points for the RASTA on SPERT project.

However, there is a considerably long queue of functions that, even for a very short time, still occupy the CPU. These must be considered because for particular tasks SPERT can be slower than a normal workstation, so these functions could increase their share of the total time. Moreover, if a considerable optimization is obtained for the most compute-intensive functions, the non-optimized will surely become more evident.

These figures should be interpretated with some caution. First, it is necessary to consider that are time percentages of the total runtime. Choosing best optimization can result in a reduction of the total runtime, and subsequentely in an increase of the percentage for some particular functions.
Besides that, it should be noted that times reported in successive identical runs may show variances because of varying cache-hit ratios that result from sharing the cache with other processes.
Moreover, quantization errors are present. The granularity of the sampling affects the results, and can also vary in successive identical runs.
Besides, care must be applied when profiling dynamically linked executables. It might be difficult to get correct profile information on the functions of a linked library. For this reason, a natural next step is to eliminate as much as possible the library functions, and analyse the differences in profile information. In the next subsection we investigate the impact of removing the library functions.

4.1.1 Dummy functions

The dummy functions idea is to substitute the library functions present within the processing with *dummy* functions, having no cost, and then to get again the profile information. We have substituted the functions calls *log*, *pow*, *exp*, *sin*, *cos*, except those calls crucial for the initialization phase. In the initialization the true functions are kept, to avoid issues of data consistency. This has been carried out only for the *sparc20* experiment described in the previous paragraph. The RASTA benchmark is exactly the same of the benchmark described before: Log-RASTA over 25539 frames. Table 4.1.1 shows the differences in running time between the case with library functions and the case with dummy functions. Times reported have been calculated using the Unix utility `time`.

PROFILING RESULTS				
routine	**sparc**	**solaris**	**acc**	**sunos**
FFT[2]	24.2	28.9	21.9	18.0
Audspec[3]	11.5	18.1	10.1	9.9
Log + Exp	9.1	6.7	7.0	3.0
Sin + Cos	8.8	7.3	5.8	11.0
Pow	5.6	5.2	4.5	22.8
MatxVect[4]	4.6	4.0	0.6	4.1
FORD[5]	4.5	5.1	6.3	3.5
fftpow[6]	4.5	6.6	3.7	1.8
fill_frame[7]	3.4	4.0	3.1	4.1
filt[8]	2.9	0	2.9	0
lpc_to_cep[9]	2.2	2.3	1.3	0.6
get_bindata	1.4	0.8	1.3	0
auto_to_lpc[10]	1.3	1.3	0.7	1.8
nl_audspec	0.9	0.5	1.0	0.6
post_audspec	0.9	1.0	0.2	0
rastaplp	0.8	0.2	0	0
rasta_filt	0.7	0.2	1.3	0
inverse_nonlin	0.3	0.2	1.3	0.6
lpccep	0.3	0.2	0.5	0
powspec	0.3	0.2	0	0
read	0	0	5.1	0

Table 4: Time percentage of the total running time for the different functions

RUN TIMES	
experiment	**time (seconds)**
solaris	42.3
sparc20	28.0
sparc	41.8
acc	46.9

Table 3: Run-times for a particular RASTA benchmark (in seconds)

We can observe a relevant difference for the *sparc20* experiment, of course due to the particular architecture used and not strictly related to the compiling options selected. Comparing the *sparc* and the *solaris* experiment, we can get an idea of the advantages obtainable using strongly optimizing compiling options: in this case it is about 1.2%. Viceversa the native compiler Sun `acc` gives poor performance compared to the others. Note that adding the options for debugging and compiling results in an additional overload of about 10 %, occurred in every experiment.

Table 4 shows the results found for every profiling experiment. The most compute-intensive functions are reported, each of them with a percentage time value. This represents the percentage of the total running time of the program accounted for by the function. These figures are a result of an average done over the data set and over the different options. However, no significant variations between different data sets were seen. Instead, as shown below, there are relevant variations between one experiment and another.

The data are ordered in decreasing order according to the *sparc* experiment, which we suppose to be the more valid and convincing of the four. The functions or routines are indicated with the names of the original C source code: in some cases, a short description is provided.

These figures indicate that overall the FFT is the most compute-intensive function. For example, considering again the *sparc* experiment, the FFT takes 24% of the total time. Note that the total sum of the time percentage is not 100%, and the remaining amount (12%) is spent in profiling activities. So we can state that the FFT occupies 28% of the total time, without profiling activities on the machine.
Other than that, it should be noted that other functions, especially library functions like *log*, *exp*, *sin*, *cos*, *pow* etc. also take a considerable amount of time. This should be taken into consideration in order to efficiently implement these routines on SPERT, probably using lookup tables. Another portion globally quite heavy is the *audspec* routine, that is the critical band analysis. From the analysis of these figures we conclude to select the FFT and the critical band analysis (*audspec*) as

PROFILING EXPERIMENTS				
Label	**Machine**	**OS**	**Compiler**	**Options**
solaris	Sparc5(icsib29)	SunOS 5.4	gcc	-DSUN4-DOS4-O2-funroll-loops
sparc20	Sparc20(anchorsteam)	SunOS 4.1.3	gcc	-DSUN4 -DOS4 -O2 -funroll-loops
sparc	Sparc5 (icsib29)	SunOS 5.4	gcc	-DSUN4 -DOS4 -O2 -funroll-loops -finline-functions -mv8
acc	Sparc5 (icsib29)	SunOS 5.4	/usr/local/lang /SC2.0.1/acc	-fast

Table 2: Compiling options used in four different experiments

of a vector architecture would not be very efficient. Another approach could be to vectorize the computation extending across several frames at the same time. For example, because the vector length characteristic for *T0* is 32, it might be opportune to consider 32 frames at once. The latter approach has the disadvantage of implying a total modification of the speech processing software architecture, therefore adding a not necessary overhead. We decided to follow the first aproach, in order to keep the problem simpler.

4.1 Profile

The first step was the analysis of RASTA code. The aim was to understand the distribution of the code on the machine, and to locate the compute-intensive sections. Once identified, these compute-intensive sections can be suitable elements to implement and optimize on the SPERT architecture.

The starting point was the original C-source code version of RASTA 2.0 running on a workstation.

A commercial RISC workstation was chosen to carry out this analysis. Several compiling options have been examined, as well as various RASTA options and input parameters. We have used a considerably large input data set, 4 Mbyte extracted from the Wall Street Journal (WSJ0) database.

Table 2 gives a description of the profiling experiments conducted: the *label* has been assigned only to distinguish the different experiments.

Table 3 shows a comparison between the run times of the RASTA-PLP processing for the four experiments described before. These result from a computation done over 25539 frames. The particular processing implemented was Log-RASTA. These figures are the result of an average taken over 10 identical runs. Run times are calculated using the UNIX utility `time`.

4 Mapping of speech front-end signal processing to high performance vector architectures

The problem of mapping the RASTA signal processing previously described on the SPERT architecture opens a number of issues, which the experiments described here attempt to address.

First of all, this work is particularly relevant because at the moment there is no efficient automatic tool for implementing algorithms on the SPERT architecture. In fact the `gcc` compiler has been ported to the *T0* processor, but it cannot generate vector instructions, and so is not capable to access any of the functionalities of the vector unit. In other words, a true native SPERT compiler is missing. Consequently, the only access to the *T0* vector unit is either through library routines or directly via the scheduling assembler. The second case is clearly more complex, but necessary when no appropriate library routines are present. In this second case, it is necessary to do the programming, the register allocation and the instruction scheduling by hand.

Besides, the SPERT architecture, whilst optimized for multi-layer perceptron tasks, should be able to perform other kinds of computation efficiently. From this point of view, the RASTA processing appears to be particularly interesting, because the involved computation is both complex and varied. For example we found the typical IIR and FIR filters, FFT (Fast Fourier Transform) and IDFT, and many mathematical functions like *sin*, *cos*, *exp*, *log*, *pow*. Besides that, there are matrix vector multiplications, reordering algorithms and loops. Other than that, there are several I/O operations (read and write from/to files). These I/O operations must be taken into account because the SPERT architecture can be much more efficient compared to a workstation for particular tasks (around one order of magnitude), but for others is surely slower. Therefore the issue of partitioning the computation, that means execute some functions on SPERT and other functions on a normal workstation, should be also considered.

Besides, the impact of having reduced input and coefficients precision should be evaluated. This is due to the conversion from a floating point architecture to a fixed point architecture. Previous studies already addressed this topic with regard to the multi-layer perceptron part of the system [10, 11], whereas this is the first time this issue is considered for the signal processing part. Saturation problems due to the fixed point arithmetic can occur, and should be carefully handled.

Another open question is how to vectorize the problem. There are two different approaches possible. The first approach is to consider the input frame as a vector, and then process it. This is a very natural and simple choice, because usually the speech processing, as described before, is done frame by frame. Note that, according to the parameters used along this experiment, a typical frame is composed of 129 samples (usually of 16 bit each). This is not the case of extremely long vectors typical of vector machines. This could tempt us to draw the conclusion that the choice

utilities.

non-linearly transformed critical band value.

This can be useful to process speech with significant additive noise. In fact the normal RASTA is effective in diminishing spectral components that are additive in the logarithmic spectral domain, as the spectral characteristics of the environment. However, uncorrelated additive noise components are not removed. These can be successfully handled by the J-RASTA processing

3.2 Typical Parameters

As said before, all the parameters can be set accordingly to the particular experiments to be carried out.
Table 1 lists the parameters used in this work, unless differently stated.

PARAMETER	VALUE
Sampling frequency	8000 Hz
Window size	20 ms
Window step	10 ms
Window points	160
FFT points	256
Critical band filters	17
Model order	8

Table 1: RASTA-PLP parameters used in all the experiments

3.3 Hardware platforms and software tools used

The hardware platforms used have been a SPARCStation 5 located at ICSI facilities, with 32MB of main memory and operating system SunOS 5.4. Later, a SPERT board has been installed within the same workstation. This SPERT board has 0.5 Mbyte of memory and a clock speed of 40 MHz.

Concerning the software tools, we used the RTL and ISA simulators for SPERT, the `gnu` compiler, assembler and debugger for SPERT, the SPERT libraries, the `gprof` utility, as well as other UNIX

introduced by frequency characteristics of the communication media.

The main steps of RASTA-PLP algorithm are listed below.

For each analysis frame:

1. compute the critical-band spectrum and take its logarithm;
2. filter the time trajectory of each transformed spectral component;
3. take the exponential function, yielding to the auditory spectrum;
4. add the equal loudness curve and raise to the power 0.33 to simulate the power law of hearing[1];
5. compute the autoregressive all-pole modeling and the cepstral coefficients.

The parameters for all the different stages can be set accordingly to the particular experiments to be carried out. Later we list the characteristic parameters used in this work.

The current implementation of the temporal filtering has chosen a fifth order FIR filter, and a first order IIR filter. The single pole of the IIR has been set to 0.94. The filter is the same for every spectral component. Note that the RASTA filter has a rather long time constant for the integration. It means that the current analysis result depends on its history, i.e. on previous outputs stored in the memory of the recursive RASTA filter. A further improvement of the RASTA algorithm, namely J-RASTA, is described in the following paragraph.

A large test has been conducted on a speaker-independent telephone digit recognition using speech that had been corrupted with convolutional noise. Experimental results from this test showed an order of magnitude improvement in error rate over conventional spectral estimation techniques.

3.1 J-Rasta

For RASTA, the bandpass filtering is done in the log domain. An alternative is to use the J-family of log-like curves:

$$y = log(1 + Jx)$$

where J is a constant that can appears to be optimally set when it is inversely proportional to the noise power, (currently typically 1/3 of the inverse noise), x is the critical band value, and y is the

[1]The equal loudness curve is an approximation to the non equal sensitivity of human hearing at different frequencies. The second operation simulates the non-linear relation between the intensity of sound and its perceived loudness

than the fixed point library, but is useful for porting applications and for calculations that are not time critical but would be difficult or time consuming to convert to fixed point.

After compilation and linking, a T0 executable is run on the SPERT board by invoking a "server" program on the attached host. The server loads a small operating system "kernel" into T0 memory followed by the T0 executable. While the T0 application runs, the server services I/O requests on behalf of the T0 process.

2.2 Simulating and profiling SPERT programs

Two simulators are available for running SPERT programs. The first of these, the ISA simulator, can be used as a replacement for a SPERT board during program debugging. It provides all the functionality of the real SPERT board (including operating system support), but at reduced speed (in the order of 100,000 cycles/second).

Alternatively, the RTL simulator includes a full register level model of the T0 chip. This results in a reduced execution speed (approx. 1000 cycles/second), but provides an exact cycle by cycle emulation of the real chip (including instruciton cache operation and interlocks). It can be used to produce traces of sections of program execution, giving an in depth view into the performance of programs at the instruction level.

3 RASTA-PLP speech analysis

The Perceptual Linear Predictive (PLP) [4] speech analysis technique is a widely used method based on the short term spectrum of speech. This technique uses some concepts of the psychophysics of hearing to derive an estimate of the auditory spectrum. However, this technique is vulnerable when the short-term spectral values are modified by the frequency response of the communication channel. The group has developed the RelAtive SpecTrAl (RASTA) [5, 9] methodology which makes PLP (and possibly also some other short-term spectrum based techniques) more robust to linear spectral distorsions.

The main idea is to replace the short-term spectrum by a spectral estimate in which each frequency channel is band-pass filtered by a filter with sharp spectral zero at zero frequency. Since any constant or slowly varying component in each frequency channel is suppressed by this operation, the new spectral estimate is less sensitive to slow variations in the short-term spectrum.

In particular, when the filtering is done in the logarithmic spectral domain, the suppressed constant spectral components reflect the effect of the convolutive factors in the input speech signal,

heart of the SPERT system is the *T0* chip. *T0* has been combined with a high speed memory system and interface circuitry to form a high performance computational subsystem that allows to improve cost-performance over a commercial general purpose processor.

Although the SPERT system is capable of performing general purpose computation, its strength is in the computationally intensive aspects of real world computing [1]. Some examples of applications for which it is suited include speech recognition, image processing and data compression. It is a good system for implementing techniques such as neural networks and digital filtering.

T0 and SPERT were developed jointly by the Computer Science Department of the University of California at Berkeley (UCB) and the International Computer Science Institute (ICSI).

2.1 Programming on SPERT

The SPERT software environment appears very similar to a conventional workstation environment.

The Torrent instruction set architecture is based on the MIPS-II instruction set, with extra coprocessor instructions added to access the vector unit functionalities. Most of the software environment is based on GNU tools. The group has ported the `gcc` C/C++ compiler, modified the `gdb` symbolic debugger to debug T0 programs remotely from the host, and enhanced the `gas` assembler to understand the new vector instructions and to schedule code to avoid interlocks. It is also possible to employ the GNU linker and other object code management utilities.

The only access to the T0 vector unit is either through library routines or directly via the scheduling assembler. There is a quite extensive set of optimized vector library routines including fixed-point matrix and vector operations, function approximation through linear interpolation, and IEEE single precision floating-point emulation. The majority of the routines are written in Torrent assembler, although a parallel set of functions have been written in ANSI C to allow program development and execution on workstations. Finally, there is a standard C library containing the usual utilities, I/O and scalar math routines.

Among the most complete libraries there is the SPERT `fxlib` library. This is a large collection of hand coded assembler routines performing a wide variety of useful functions. `fxlib` includes memory operations, integer and floating point vector arithmetic, fixed point matrix/vector arithmetic, function approximation and Fourier transforms. The `fxlib` routines use the MIPS C calling conventions and consequently can be called from both C and C++. Portable versions written in C are also available, allowing development of SPERT applications on workstations.

There is also an optimized IEEE floating point matrix/vector library. This is considerably slower

1 Introduction

A phoneme-based speaker independent continuous speech recognition system has been developed by the Realization Group at ICSI. Acoustic information is first processed by a features extractor. The system then utilizes a MLP (Multilayer Perceptron) to generate emission probabilities for a hidden Markov model (HMM) speech recognizer [7].

The features extraction, accomplished by appropriate signal processing algorithms of high complexity, represents the front-end of the system and therefore is a crucial stage for speech recognition systems. The capability of operating in adverse conditions, with high background noise and different channel characteristics, is one of the major goals when developing automatic speech recognition systems for use in real world environments. In addition, the need for real-time performance and the large amount of computation often require the use of specialized architectures.

The Torrent architecture [2], developed at ICSI, is a high performance vector architecture optimized for neural networks and signal processing tasks. This paper is aimed to study the mapping of the front-end part of speech recognition systems - a robust algorithm called RASTA-PLP (RelAtive SpecTrAl - Perceptual Linear Predictive)[5] - to this architecture.

Currently the speech recognition system is partially implemented on SPARCstations and partially on the RAP [6] (Ring Array Processor), another architecture previously developed by the group. The long term aim is to implement the whole system on the new Torrent architecture. The work described here is part of such large-scale effort.

Section 2 describes the characteristics of the Torrent architecture and its first implementation in form of the SPERT board. This is followed in Section 3 by an analysis of the speech processing algorithms. Finally, in Section 4 we discuss how to map a speech processing task to the system and compare the resulting performance.

2 Torrent, T0 and SPERT

Torrent is a vector architecture. It is based on the industry standard MIPS RISC microprocessor with no-floating point unit and an additional fixed point vector unit.
T0 (Torrent0) is a single chip fixed point vector microprocessor, the first implementation of the Torrent architecture. It can execute up to two operations per-cycle on 8 word vectors, or in other words, compute 16 results in a single cycle. For more details concerning internal operation, performance and interfaces, see [3].

SPERT is a low cost neural network and signal processing accelerator board for workstations. The

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Mapping of speech front-end signal processing to high performance vector architectures

Paola Moretto

TR-95-063

December 1995

Abstract

Front-end signal processing is a crucial stage for speech recognition systems. The capability of operating in adverse conditions, with high background noise and different channel characteristics, is one of the major goals when developing automatic speech recognition systems for use in real world environments. In addition, the need for real-time performance and the large amount of computation often require the use of specialized architectures. We describe the study of the mapping of a fundamental part of speech recognition systems - a robust speech front end algorithm called RASTA - to the *Torrent* architecture. This architecture, developed at ICSI, is a high performance vector architecture optimized for signal processing task. The mapping problem is particularly relevant because at the moment there is no efficient automatic tool for implementing algorithms on the Spert architecture. An appropriate algorithms analysis is shown, as well as the design of optimal library routines which allow to fully exploit the vector architecture. Preliminary functional and performance comparisons with more standard architectures are shown.