

FreeTTS - A Performance Case Study

Willie Walker
Paul Lamere
Philip Kwok

Sun Microsystems Laboratories
Burlington, Massachusetts

{William.Walker, Paul.Lamere, Philip.Kwok}@sun.com

1. Introduction

The Java™ platform has a stigma of being a poor performer and has often been shunned as an environment for developing speech engines. We decided to better understand this stigma by writing a speech synthesis engine entirely in the Java programming language.

Remarkably, we were able to get better performance from our engine than an engine using similar algorithms written in C. Our team, which is composed of three engineers with significant backgrounds in the C and Java programming languages, also found it easier to make algorithm modifications in the Java programming language than in C. In addition, we were able to create an engine with more flexibility and our engine generated intelligible speech only four weeks after we wrote our first line of code.

Our work, named FreeTTS, is based upon two speech synthesizers: the Festival Speech Synthesis System [festival2001], and Flite [flite2001]. Festival, which Sun Microsystems funded through collaborative research, is an extremely flexible speech synthesis research environment written using Scheme and C++, with no particular attention paid to performance. Flite (Festival Lite) was written at Carnegie Mellon University (CMU), and is based upon Festival. Flite is written entirely in C with great detail paid to size and performance on embedded platforms. The size and performance requirements of Flite, however, drastically reduce its flexibility. To get the best of both worlds, we based FreeTTS's algorithms on Flite, but the architecture on Festival.

In this document, we briefly describe the overall process FreeTTS uses to convert general text into synthesized speech. We then describe the performance issues raised by such a system and how we addressed those issues.

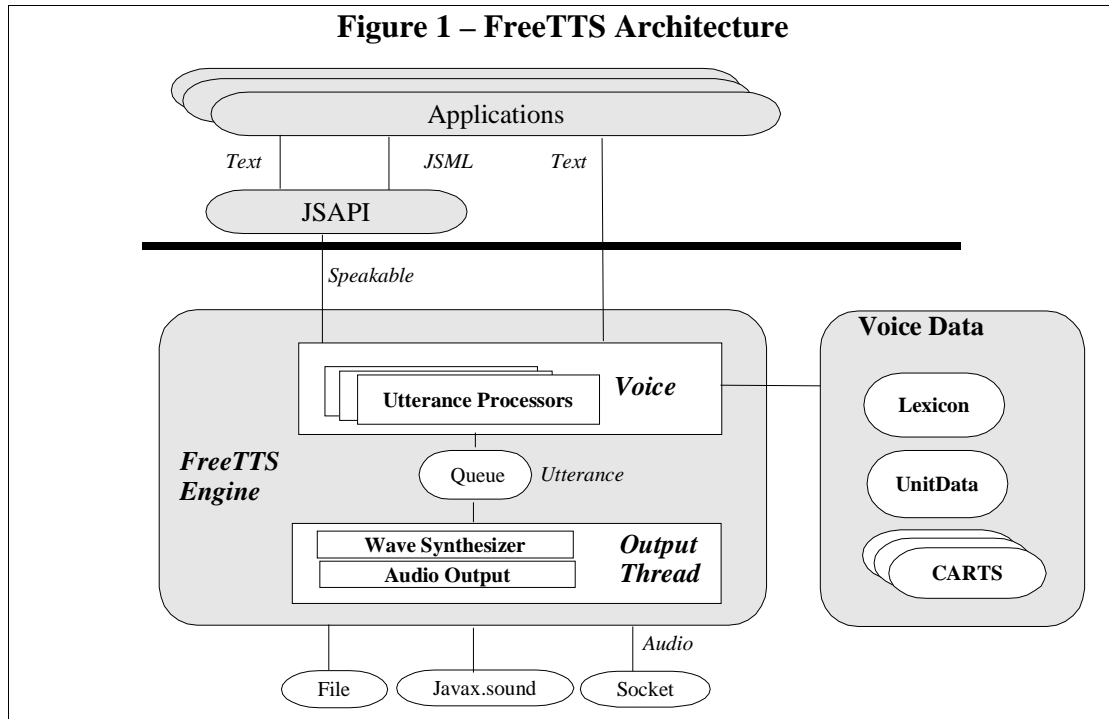
Summary of the Synthesis Process

To synthesize speech, FreeTTS breaks the input text into sets of phonemes¹ and then converts those phonemes into audible speech. FreeTTS does this by performing successive operations on the input text. FreeTTS stores the the cumulative results of each operation in an utterance structure that holds the complete analysis of the text.

Figure 1 shows the overall architecture for FreeTTS. The core of FreeTTS is an engine that contains a voice and an output thread. The voice consists of a set of utterance

¹ A phoneme is a member of the set of the smallest units of speech that serve to distinguish one utterance from another in a language or dialect.

processors that create, process, and annotate an utterance structure. Associated with the voice is a data set that is used by each of the utterance processors. The output thread is responsible for two actions: synthesizing an utterance into audio data and then directing this data to the appropriate audio playback mechanism.



The Voice and Utterance Structures

The heart of FreeTTS lies in its voice and utterance structures. The voice maintains the global information about the synthesis process: the locale, the pronunciation lexicon, the unit database,² and the wave synthesizer. The voice also maintains the set of utterance processors used to create and annotate the utterance structure.

The utterance structure is a temporary object the voice creates for each audio wave it generates. The voice initializes the utterance structure with the input text and then passes the utterance structure to a set of utterance processors in sequence. Once the input text is processed (e.g., sent to an audio output device), the voice discards the utterance structure.

Each utterance processor adds additional items to the utterance structure in an hierarchical and relational manner. For example, one utterance processor creates a relation in the utterance structure consisting of items holding the words for the input text. Another utterance processor creates a relation that consists of items describing the syllables for the words, with each syllable item *pointing back* to the individual word items created by the other utterance processor.

By arranging the utterance structure using relations, the utterance processors can perform sophisticated relational queries on the utterance structure. An example query text is

² A 'unit' is a single portion of speech which may range in size from a whole phrase down to a phoneme.

"R:SylStructure.parent.parent.word_numsyls." Reading from right to left, this means "find the number of syllables in the word that's the parent of the parent of the syllable relation for the particular item of interest." These types of queries are used throughout FreeTTS by the various utterance processors.

Synthesis Steps

There are a number of steps in the synthesis process. Many of these steps need to be customized depending upon the locale and the type of synthesis employed. A typical FreeTTS voice will perform the following steps to convert written text to speech:

- **Text Normalization** – Performed via an utterance processor that converts the input text into a stream of words. For example, the text “Dr. Smith lives on 33 Garden Dr.” would be converted to “doctor smith lives on thirty three garden drive.” The text normalization process deals with a wide variety of cases including numbers, dates, times, titles, and place names.
- **Linguistic Analysis** – Performed via an utterance processor that determines semantic information such as phrasing and part-of-speech information.
- **Lexical Analysis** – Performed via an utterance processor that determines the pronunciation, syllable identification, and stress for each word of the utterance. Typically, FreeTTS will use a lexicon to determine this information. If a word is not in the lexicon, however, FreeTTS falls back to a set of sophisticated letter-to-sound rules.
- **Prosody Generation** – Performed via an utterance processor that determines parameters for pauses, pitch, duration, tone, stress, and amplitude. These processors will typically use classification and regression trees (CARTS) [brieman84] to generate this prosody information.
- **Speech Synthesis** – Generates audio data, typically by concatenating speech units based on diphones or other units of speech. Synthesis can be particularly CPU intensive since it involves a great number of floating point operations.

When broken down into these discrete steps, the synthesis process is relatively straightforward. The process, however, pushes performance bounds in two dimensions. First, there are large data sets to contend with, with the lexicon and the unit database comprising the largest amount of data. Second, there is a significant amount of floating point computation involved. Not surprisingly, the audio synthesis processing takes the longest of all the steps.

In the following sections, we describe the performance issues in more detail, and also describe how we addressed those issues in FreeTTS.

2. Execution Time Base Lines

For our first pass at FreeTTS, we decided to keep everything as simple as possible. Our primary concentration was to design the overall architecture and just get it to talk. As a result, we did not pay much attention to performance. For example, instead of concentrating on the time to load the lexicon and diphone database, we just parsed the data sets from separate ASCII files. After our wonderful experience of hearing FreeTTS speak for the first time, we then examined the performance of the engine.

Our first performance work on FreeTTS involved reducing the execution time in two dimensions: the first was the time it took to load the data sets, and the second was the time it took FreeTTS to synthesize speech once it was given text. Before doing any performance work, however, we took a number of baseline measurements.

Data Set Loading Baseline

FreeTTS's C-based counterpart, Flite, does not load the data sets it uses from files. Instead, Flite defines all of its data as static constants directly in the source code. As a result, the data and code are loaded simultaneously, and thus Flite's effective load time is zero. FreeTTS, however, loads its data sets from files, so we gathered metrics on how long it took FreeTTS to load the data sets. Table 1 shows the load times for the larger data sets used by FreeTTS.

Data Set	Bytes	Load Time (seconds)
All CARTS	36,496	0.41
Lexicon	1,705,833	6.25
Letter to sound rules	261,318	0.65
Unit database	7,828,514	13.76
Total	9,832,161	21.07

Table 1 - Initial Data Set Load Times

As can be seen by the table, the lexicon and unit database took a significant amount of time to load.

Execution Speed Baseline

We decided that if we could not make FreeTTS run quickly, then other performance improvements would not be worth the effort. Before working on the execution speed, we established a performance baseline comparing the processing speed of FreeTTS to Flite. For the baseline, we ran both the Flite and FreeTTS synthesis engines using the "Alice" test: a large input text consisting of an abbreviated version of Lewis Carroll's "Alice in Wonderland." The input text contains 4293 words, with an approximate total speaking time of nearly 22 minutes. Table 2 shows the timings in seconds for each of the utterance processors when run on a single SPARC® processor (v9) operating at 296 MHz with 128MB. All FreeTTS classes and data were directly in the CLASSPATH (i.e., JAR³ files were not used).

³ The Java™ Archive (JAR) file format allows multiple files, including class files and related resources, into a single archive file.

Processor	Flite (secs)	FreeTTS (secs)	FreeTTS:Flite (ratio)
Tokenization	0.216	0.393	1.819
Normalization	0.260	0.801	3.081
Phrasing	0.537	0.371	0.691
Segmentation	2.592	1.349	0.520
Pause Identification	0.060	0.083	1.383
Intonation	7.704	10.401	1.350
Post Lexical Analysis	0.838	0.777	0.927
Duration	7.749	6.860	1.350
F0 Contour	10.350	11.119	1.074
Unit Selection	1.058	1.576	1.490
Wave Synthesis	11.206	47.399	4.230
Total	42.570	81.129	1.906

Table 2 - Initial Flite and FreeTTS Processing Metrics

From this table, it can be seen that our first version of FreeTTS spent a significant amount of time synthesizing the wave data. This time contributed greatly to the result of FreeTTS being nearly twice as slow as Flite.

3. Data Set Loading Improvements

We found it unacceptable that FreeTTS took 21 seconds to load the data sets. Although this was a one-time penalty to incur when running FreeTTS, we would frequently start and stop FreeTTS during the development phase. As such, any improvements we could make in this area would directly benefit us by reducing development turnaround time.

For our first attempt at reducing the load time, we experimented with defining the data as `static final` constants directly in the code, but uncovered some unexpected results:

- The Java™ virtual machine⁴ limits the constant pool to 64K entries per class [lindholm97]. In addition, the amount of code per class is also restricted to 64K bytes. As a result, we needed to break the larger data sets into several classes.
- The resulting class files were much larger than expected. This is due to the fact that when an array of primitive types is defined in an application written using the Java programming language, the compiler generates a sequence of code that initializes each member of the array. For example the code:

```
private static float[] floatArray = {
    1.0f, 2.0f, 3.0f, 4.0f, 5.0f
};
```

⁴ As used in this document, the terms "Java™ virtual machine" or "JVM™" mean a virtual machine for the Java platform.

Results in code that looks like this:

```
newarray float # allocate the array
dup
iconst_0      # store the first value
fconst_0
fastore
dup
iconst_1      # store the second value
fconst_1
fastore
etc.
```

Thus, each element of the array consumes four bytes for the value itself, plus four bytes of code to store the value in the array, plus a four byte array index. This results in three times the amount of space required to initialize the array as compared to the native-C counterpart.

- It took longer to load the resulting class files than it took to read and parse the data from a file. It took longer not only because there were three times the amount of data to read, but also because the byte code verifier [lindholm97] needed to check a large amount of extra code.

As a result, we abandoned these experiments early and instead focused on alternative methods for loading our data sets.

Lazy Tokenization

When analyzing a run of FreeTTS, we determined that much of the load time was spent parsing the ASCII data sets. To experiment with performance improvements in this area, we chose to delay as much parsing of the data until absolutely necessary.

For example, an entry in the lexicon data set looks like the following:

```
abdication0 ae1 b d ih k ey1 t ih ng
```

Each word is encoded with its part of speech and is followed by a list of phonemes. Our initial pass at loading the lexicon completely parsed each line as it was read: the combined word and part of speech became a key for a `HashMap` and the phoneme list was stored as an array of strings.

In an attempt to improve the load time, we did not parse the list of phonemes when they were first read in. Instead, we saved the list as one string, and only parsed it when the phoneme list for a word was requested. As seen in Table 3, the results were promising.

Method	Times in seconds	
	Load Time (secs)	Lookup Time (secs)
Tokenize on Load	6.3	1.1
Tokenize on Lookup	2.7	1.1

Table 3 - Comparison of Tokenization Strategies

In this table, the times represent the total time FreeTTS spent looking up words for the "Alice" test. As can be seen, lazy tokenization of the lexicon allowed us to cut the load time by more than one-half without a noticeable degradation in lookup time.

Binary Data

Our next attempt at reducing load time involved converting the ASCII files into a binary form. Representing the data sets as ASCII files is extremely inefficient: not only does it take more disk space, but it also requires computation time to turn the text into the appropriate format. For example, the diphone database consists mostly of numerical data. When the diphone data is represented as an ASCII file, loading it requires parsing over 1.8 million strings, only to turn those strings into their appropriate primitive data types.

We assumed that by reading the primitive types in directly, we would eliminate all the overhead caused by parsing the data strings before converting it into primitive types. Because the diphone database was the largest, we focused our efforts on that first. As can be seen in Table 4, our assumption proved to be correct for the diphone database. We also tried the new IO (nio) package for loading binary data, and it provided the best performance overall.

File Format	Load Time
ASCII	13.760s
Binary	1.491s
New IO Binary	1.083s

Table 4 - ASCII vs. Binary Load times

Given the success of using a binary format for the diphone data, we revisited the lexicon to see if we could get similar performance. Unfortunately, we had only a negligible improvement in performance. Upon further analysis, we discovered during the loading of the lexicon, the Java virtual machine was spending much of its time creating strings and populating the `HashMap` that we were using to hold the lexicon. This time far overshadowed the improvement we were getting from using a binary form for the data. As a result, we decided to refrain from using a binary format for the lexicon.

JAR Files

Finally, we were concerned that placing the binary data sets in a JAR file would be detrimental to the loading performance. As can be seen in Table 5, this is true to an extent:

Loading Method	Load Time
Raw Classes and binary files	3.4s
Uncompressed JAR Files	5.1s
Compressed JAR files	7.6s

Table 5 - Comparing Different Methods of Loading FreeTTS

Data Set Loading Summary

There are several other performance improvements we could make to load the data. For example, we could use a different data structure than a `HashMap` to store our lexicon. Given our success in reducing the total load time from 21 seconds down to under 4 seconds, however, we decided to focus on lowering the execution time.

4. Execution Time Improvements

Computing the audio output for the "Alice" test initially took over 47 seconds. This compared unfavorably with Flite which took less than 12 seconds to perform the same task. Profiles of this step showed two areas of possible optimization:

1. Excess time was being spent performing buffer copies.
2. A large amount of time was spent in an inner loop calculation.

Eliminating Buffer Copies

Due to an architectural decision, the audio output classes expect data in the form of `byte` arrays, but the audio wave synthesizer generates data as `short` arrays. In our first implementation, we wrote the audio wave synthesizer data to an in-memory `ByteArrayOutputStream` wrapped in a `DataOutputStream`, and then finally normalized the data in yet a third `byte` buffer. By modifying the wave synthesizer to generate `byte` data directly (a simple modification), we eliminated two buffer copies. This reduced the wave synthesis time for "Alice" by 13 seconds (from 47 to 34), a significant improvement.

Optimizing the Inner Loops

The final stage of the synthesis process generates the audio samples using a linear predictive coding (LPC) algorithm. The LPC algorithm generates an output sample by filtering the last 10 sample outputs with a set of filter coefficients associated with the sample frame, resulting in 22 floating-point operations per frame. As a result, our "Alice" test case requires over 225 million floating-point operations to generate the audio data.

Through experimentation, we found that array indexing was responsible for a considerable fraction of the time spent in the inner loop. Because the Java programming language specifies that all array indexes are checked at run time, array accesses are potentially more expensive in the Java programming language than their counterparts in C.

By maintaining the output buffer in a custom linked list instead of an array, we were able to eliminate all array indexing from the inner loop. This reduced the wave synthesis time by 12.5 seconds (from 34 seconds to 21.5). As mentioned later in this document, we discovered that optimizing compilers such as the Java HotSpot™ server compiler would have obviated the need to move from arrays to linked lists.

Other minor optimizations reduced the total wave synthesis time by another 1.1 seconds. We decided we were reaching the point of diminishing returns and went on to other areas for improvement.

Utterance Structure Modifications

Aside from the computation of the wave, we also found that FreeTTS spends a lot of time traversing the utterance structure. As mentioned previously, subsequent utterance processors refer to the results of previous utterance structures, and they do so using a

query text (e.g., "R:SylStructure.parent.word_numsyls"). Most of the queries are relational, and typically refer to the item just before or after a given item.

Our initial implementation of the utterance structure stored utterance items using a `java.util.List`, so searching for the item before or after a given item always resulted in a linear search. We redesigned the storage of our utterance items to use a linked list, with the end result being a savings of 4 seconds in the overall processing time of "Alice."

Algorithm Improvement

Up to this point, our performance improvements concentrated solely on making our code run faster, but still kept the algorithms identical to Flite. Although the basic algorithms were identical to Flite, we were able to take advantage of some of the high performance data structures such as the `HashMap` provided by the Java platform. Our improvements were actually quite successful: for the "Alice" text, we were able to match the overall total processing time of Flite, including the time it took to load the FreeTTS data sets.

Although we realized there were more performance improvements we could make without changing the algorithms, we decided there were some algorithmic changes we could make that would have a significant benefit.

Studying the data from Table 2, we turned our attention to the intonation processor. Through a simple process of elimination, we discovered that the relational query text continually used throughout FreeTTS was being parsed each time it was used. Because the query text is static (there are over 600 instances in the data sets), we looked at preprocessing the text just once, and then using the processed form in subsequent queries. The results were dramatic: preprocessing the query text shaved over 12.5 seconds off the total time to process the "Alice" text. In addition, the results were nearly identical whether we preprocessed the query text at load time or if we waited until the query was run the first time. Since the same query text was used by a number of utterance processors, by improving its performance we not only improved the intonation processors, but many other processors as well.

We considered making a similar change to Flite to compare the results. We refrained from doing so because it would have required too much work (e.g., the manual garbage collection required). Thus, we saw another benefit to using the Java programming language: it allows us to make significant algorithm changes faster and easier.

A New Java™ 2 Platform Release

During the development of FreeTTS, the Java team released several beta versions of the Java™ 2 Platform, Standard Edition (J2SE™) version 1.4. As each new version was released, we tested FreeTTS to measure its performance under the new release. We were pleasantly surprised to find significant performance increases just by upgrading to a new release. For instance, when upgrading to the beta2 release J2SE™ 1.4, the time to synthesize wave data for the "Alice" text dropped from 21 seconds to 14 seconds. (The total time dropped from 34 seconds to 24 seconds). Imagine our mixed emotions after working hard for several days to shave two seconds off the time, but then have 10 seconds eliminated just by a beta upgrade.

One notable improvement in the J2SE 1.4 upgrade was improved range check elimination. The Java HotSpot compiler can detect certain array access idioms, particularly `for` loop accesses, and eliminate the range checking on the array index if it determines the index always falls within range of the array. Some of the aforementioned optimizations that we performed on FreeTTS to eliminate array accesses to avoid index range checking became unnecessary because of these Java HotSpot compiler improvements.

Improving Performance with the Java HotSpot™ Virtual Machine

The Java HotSpot virtual machine is Sun Microsystems' virtual machine for the Java platform. The J2SE 1.4 release provides two flavors of the Java HotSpot virtual machine: a client compiler that provides for faster program start times, and a server compiler that maximizes program speed but with a longer program start time and a larger memory footprint. The server compiler performs a wide range of optimizations including aggressive inlining of virtual methods, loop unrolling, dead code elimination, common sub-expression elimination, and array range check elimination.

Speech applications are often constructed as client/server applications with the recognition and synthesis engines running as separate servers possibly on separate machines. This architecture can improve the scalability, flexibility and reliability of a system. With this in mind, we developed a client/server version of FreeTTS that allows the synthesis engine to receive synthesis requests via a socket connection, synthesize the wave data and return it to the client via the socket. When used in contexts such as this where startup time is less important than overall TTS performance, FreeTTS can be run using the server compiler with a significant performance boost. Table 6 shows a comparison of Flite and FreeTTS running with the client and the server compiler.

1-CPU 296 MHz SPARC® processor (v9) w/ 128Mb			
	Load Time	Run Time	Total
Flite	0s	44.3s	44.3s
FreeTTS -client	3.4s	24.4s	27.8s
FreeTTS -server	5.9s	32.7s-41.7s	38.6s-47.6s

Table 6 - Comparison of Client and Server compilers

We were puzzled by these results. First of all, the timings with the server compiler were inconsistent, ranging from 38 to 47 seconds. Secondly, the performance of the server compiler was worse than the client compiler. Some investigation showed that the Java HotSpot server compiler needs a longer period of time to identify and compile the hot spots. We ran the test again replacing the “Alice” input text with the text of Jules Verne's *Journey to the Center of the Earth*, which is about 20 times as long. Table 7 shows the results of this test.

1-CPU 296 MHz SPARC® processor (v9) w/ 128Mb			
	Load Time	Run Time	Total
Flite	0s	955.4s	955.4s
FreeTTS -client	3.3s	505.5s	508.8s
FreeTTS -server	5.8s	347.0s	352.8s

Table 7 - Client/Server comparison with longer input

Giving the server compiler a longer period of time to optimize the hot spots proved beneficial: FreeTTS using the server compiler is about 30% faster than running it using the client compiler.

Multiple CPU Improvements

Flite is a single threaded application and as such cannot take full advantage of a multi-CPU system. The ease in which threads can be created using the Java programming language, however, permitted us to quickly make FreeTTS a multi-threaded application. With one thread producing utterances and a second thread generating wave data from these utterances, the virtual machine can distribute these threads among the available CPUs to achieve a further performance boost.

Table 8 shows the results of processing the “Journey” text on a 2-CPU 360 MHz SPARC® processor (v9) with 512 Mb of memory⁵.

2-CPU 360 MHz SPARC® processor (v9) w/512 Mb			
	Load Time	Run Time	Total
Flite 2-CPU using 1 CPU	0s	803.2s	803.2s
Flite 2-CPU using 2 CPUs	0s	800.4s	800.4s
FreeTTS -server 2-CPU using 1 CPU	5.2s	282.9s	288.1s
FreeTTS -server 2-CPU using 2 CPUs	3.1s	193.1s	196.2s

Table 8 - Single CPU/ Multi-CPU Performance Comparison

This table shows that, as expected, Flite achieves nearly identical run-times when running on a single or multi-CPU system. FreeTTS, however, shows a 33% improvement in runtime when running on a 2-CPU system.

Time-to-First-Sample Performance Tuning

A critical benchmark in text-to-speech synthesis engines is the time from when the synthesizer receives the text to synthesize to the time the first audio sample is generated. This benchmark is called the *time-to-first-sample*.

FreeTTS optimizes the time-to-first-sample in two ways. First, it partitions the final wave synthesis and wave output steps into a separate thread. This allows utterance generation to occur concurrently with wave synthesis and audio output. Since samples can be synthesized much faster than they can be output as audio, this partitioning allows utterance processing to be overlapped with the final audio output time, reducing the

⁵ The Solaris™ 8 Operating Environment command, `psrset`, was used to limit the run to a single CPU for the 1-CPU results.

overall processing time. Additionally, since the audio wave synthesis is in a separate thread, the Java virtual machine can potentially allocate the synthesis thread to its own CPU, boosting overall performance.

Second, FreeTTS allows audio output to be streamed such that as soon as the first sample of audio is generated, it is sent to the audio system to be played. This is unlike Flite which generates the entire wave output for an utterance before sending the resulting data to the audio system to be played.

Table 9 compares the *time-to-first-sample* for Flite and FreeTTS while running on single CPU 296 MHz (1-CPU) system and a dual CPU 360 MHz system (2-CPU).

Input Size (Words)	Flite		FreeTTS -client		FreeTTS -server	
	1-CPU Time	2-CPU Time	1-CPU Time	2-CPU Time	1-CPU Time	2-CPU Time
1	13ms	11ms	5ms	4ms	5ms	3ms
2	22ms	18ms	8ms	6ms	11ms	6ms
5	40ms	34ms	18ms	14ms	27ms	14ms
10	79ms	68ms	38ms	30ms	50ms	26ms
100	1034ms	813ms	864ms	690ms	631ms	485ms

Table 9 - Time-to-first-sample

As with other performance improvements, we attempted making similar changes to Flite but found that the C programming language made this a very difficult task.

Execution Time Summary

With all of the performance improvements described previously, we were able not only to improve the performance of FreeTTS, but also to make FreeTTS run *considerably faster* than Flite. Table 10 shows a comparison of utterance processing times of the “Journey” text for Flite and FreeTTS running on single CPU and multiple CPU systems.

- **1-CPU** - Single CPU, 296 MHz SPARC® processor (v9) with 128 MB memory
- **2-CPU** - Dual CPU, 360 MHz SPARC® processor (v9) with 512 MB memory

Process	Flite 1-CPU	Flite 2-CPU	FreeTTS 1-CPU	FreeTTS 2-CPU
Tokenization	4.444	3.981	8.219	6.211
Normalization	5.493	4.665	8.359	6.750
Phrasing	18.497	15.476	3.255	1.646
Segmentation	19.415	16.921	18.697	13.026
Pause Identification	7.607	6.340	0.981	1.002
Intonation	175.041	145.803	24.500	17.152
Post Lexical Analysis	19.706	16.159	2.336	1.167
Duration	183.933	153.418	36.041	28.370
F0 Contour	257.752	214.969	39.749	29.194
Unit Selection	26.097	23.897	35.250	34.167
Wave Synthesis	203.002	189.633	167.620	54.979
Total	920.990	791.265	345.050	193.723

Table 10 - Processing “Journey” with the Java HotSpot™ server compiler (times in seconds)

Our original performance goal was to make FreeTTS run as fast as Flite. However, with only minor algorithm changes, FreeTTS can now run more than 4 times faster than its native-C counterpart.

5. Memory Footprint Analysis

Since our primary goal in developing FreeTTS was to investigate the performance characteristics of a text-to-speech engine written in the Java programming language, we were primarily concerned with the previously discussed performance metrics of overall processing time and the time-to-first-sample. We spent very little effort trying to reduce the memory footprint of FreeTTS. As such, Flite currently has a much smaller memory footprint than FreeTTS.

6. Helpful Features of the Java™ platform

During the development of FreeTTS, we increasingly appreciated the many features of the Java platform that made our job as programmers easier.

- **Object Oriented Language** - Object-oriented languages are well suited to support the 'pluggable' and 'configurable' requirements of speech synthesis engines.
- **Dynamic Loading of Code** - Since the Java programming language allows code to be loaded dynamically at runtime, extending the speech synthesis engine with new voices is easy. As a result, voices can be packaged into JAR files and deployed separately from the engine.
- **Multi-threaded Language** - We were able to easily add multithreading to FreeTTS, resulting in faster execution speeds and reduced time to first sound.
- **Garbage Collection** - Since we did not have to explicitly manage memory, writing code was easier and the resulting code was less complex.

- **Vast API** - The large set of APIs that are part of the J2SE platform made developing FreeTTS much easier. In addition, because J2SE provided high-quality implementations of data structures and algorithms, we were able to get a significant performance boost "out of the box." In particular, the collections API, regular expressions API, and the Java Sound API were much appreciated.
- **Portability** - We are able to use our FreeTTS JAR files on the Solaris™ 8 Operating Environment, Windows, and Linux systems without making any changes. This compares favorably to Flite which must be ported and built specifically for each supported environment.
- **Documentation** - By conforming to the Javadoc™ documentation standards when writing FreeTTS, we were able to generate high-quality API documentation for the FreeTTS synthesis system directly from the source code.

7. Summary

When we started our study of the performance characteristics of a speech synthesis engine programmed in the Java programming language, our expectations were that it would hopefully be able to run nearly as fast as the native-C counterpart. Through using some straightforward optimizations and relying on the aggressive optimizations performed by the Java HotSpot compiler, we were pleased to find that FreeTTS runs two to four times faster than its native-C counterpart, Flite.

Clearly, it would be possible for us to roll some of these optimizations back into Flite with the likely result of improving Flite's performance to levels similar to FreeTTS. The lack of Java platform features such as garbage collection and high-performance collection utilities, however, makes performing these optimizations in Flite much more time consuming from a programming point of view.

Acknowledgments

The authors of this document would not have been able to create FreeTTS if Alan Black at CMU had not written the Flite synthesis engine. We greatly appreciate his work and his depth of knowledge in this area. Furthermore, we thank our management for allowing us to tackle this problem.

References

- [black96] Black, A., and Hunt, A., 1996. "Generating F0 Contours from ToBI Labels Using Linear Regression." ICSLP96, Volume 3, pp 1385-1388.
- [brieman84] Breiman, L., Friedman, J. H., Olshen, R.A., and C. J. Stone. 1984. "Classification and Regression Trees." Wadsworth, Belmont, CA.
- [cmulex] "The CMU Pronouncing Dictionary," Version 0.6. Unpublished content available at <http://www.speech.cs.cmu.edu/cgi-bin/cmudict/>.
- [festival2000] Black, A., Lenzo, K., 2000. "Building Voices in the Festival Speech Synthesis System Processes and issues in building speech synthesis voices Edition 1.2:beta, for Festival version 1.4.1" Unpublished content available at <http://www.festvox.org/festvox>.
- [festival2001] Black, A., Taylor, P., and Caley, R., 2001. "The Festival Speech Synthesis System, Version 1.4.2." Unpublished document available via <http://www.cstr.ed.ac.uk/projects/festival.html>.
- [flite2001] Black, A., and Lenzo, K., 2001. "Flite," Version 0.91. Unpublished source code available via <http://www.cmuflite.org/>.
- [hunt89] Hunt, M., Zwiernyski, D., and Carr, R., 1989. "Issues in High Quality LPC Analysis and Synthesis." Eurospeech89, Volume 2, pp 348-351.
- [lindholm97] Lindholm, T., and Yellin, F., 1997. "The Java™ Virtual Machine Specification." Addison Wesley, Section 4.10.
- [ostendorf1995] Ostendorf, M., Price, P., Shattuck-Hufnagel, S., 1995. "The Boston University Radio News Corpus." Technical Report ECS-95-001, Electrical, Computer and Systems Engineering Department, Boston University.
- [silverman1996] Silverman, K., Beckman, M., Petrelli, J., Ostendorf, M., Wightman, C., Price, P., Pierrehumbert, J., and Hirschberg, J., 1996. "ToBI: A standard for labeling English prosody." Proceedings of ICSLP 92, Volume 2, pp. 867-870.

About the Authors

Willie Walker joined the Sun Microsystems Laboratories in 1998 and became the principal investigator for the Speech Group in 2000. Willie's currently helping create a voice architecture for Sun and is also leading the migration of the Carnegie Mellon University Sphinx speech recognition system to the Java platform. Willie also led the effort to create FreeTTS. Before joining the Speech Group, Willie worked on the Java Foundation Classes. His major contributions to that effort include the design and implementation of the Java Accessibility API and the Multiplexing Look and Feel for Swing. Prior to joining Sun, Willie worked for Digital Equipment Corporation on the X Window System and Motif. His work on X/Motif included the XKB keyboard extension as well as investigating making the X Window System more accessible to people with disabilities. While there, he also helped found DACX, the Disability Action Committee for X.

Paul Lamere has been writing software for the last twenty years. He spent much of that time developing real-time, embedded systems. His software can be found embedded in medical instrumentation, manufacturing equipment, and even the U2 spy plane (although he can't talk about that too much). Paul switched over to developing in the Java programming language about five years ago and he hasn't looked back since. Paul joined the Sun Microsystems Laboratories Speech Group in the summer of 2000. His current software interests are developing speech applications, speech engines, and extreme programming.

Philip Kwok joined the Sun Microsystems Laboratories Speech Group in the summer of 2000. He graduated with a degree in Computer Science from Hampshire College in May 2000. His thesis involved building a network visualization system for the SSF Network. He was previously employed as research intern at the DIMACS Research Center where he carried out large-scale networks simulation research. There, he built the first Java platform version of the SSF Net, and implemented the OSPF routing protocol on top of it. He also interned at the IBM TJ Watson Research Center, where he worked on a software library system for internal use.