Architectural Optimizations for Text to Speech Synthesis in Embedded Systems

Soumyajit Dey, Monu Kedia, Anupam Basu

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India - 721302

Abstract— The increasing processing power of embedded devices have created the scope for certain applications that could previously be executed in desktop environments only, to migrate into handheld platforms. An important feature of the computing systems of modern times is their support for applications that interact with the user by synthesizing natural speech output. Such applications deliver state of the art performance in desktop environments. However, the real-time performance of such applications in handheld platforms with on-line incoming text streams have not been explored till date.

In this work, the performance of a Text to Speech Synthesis application is evaluated on embedded processor architectures and modifications in the underlying hardware platform are proposed for real time performance improvement of the concerned application.

I. Introduction

In recent years, embedded computing has been growing tremendously in both popularity and complexity. Increased processing power, technology scaling and availability of heterogeneous design platforms like FPGAs, multi-core processors,custom ASICs, CPLDs etc. have resulted in powerful handheld devices that complement and accelerate human actions in almost every sphere of modern society. This philosophy of portable computing requires novel interface technologies that increase the effectivity of such platforms. Text to Speech (TTS) conversion is one such application which has the potential to enhance the Human-Computer Interaction (HCI) capabilities of portable devices. TTS engines are integral parts of desktop applications like *Screen Reader*, assistive software for the physically challenged etc. However, such interfacing technologies have much higher implication in the domain of portable and pervasive devices.

In real life situations, an embedded handheld device may require to handle on-line incoming text and synthesize the output speech in real-time. However, the performance of such speech synthesis engines have been found to be lacking in the context of low power embedded processors, given a situation where the processing has be done on-line with high throughput requirement. This situation calls for architectural optimizations while respecting the constraints of embedded processing like low-power Processing Elements (PEs), small on-chip memories, moderate bus speed etc.

This work explores such domain specific architectures for speech synthesis by exploiting application characteristics like availability of concurrency, scope of custom hardware and memory demands.

The paper begins with a brief introduction to relevant speech synthesis techniques as well as an introduction to *Shruti* [1], the concatenative speech synthesis engine used in this work in Section II. Section III provides a brief review of relevant architecture design works. Section IV describes an initial performance analysis of the application which provides an insight into the memory demands and bottlenecks. Section V reports the gradual architectural refinement steps driven by design decisions based on the performance analysis and initial refinements which finally lead to the conclusions and planned future directions in Section VI.

II. Text to Speech Synthesis

Speech synthesis involves the algorithmic conversion of input text data to speech waveforms. Speech synthesizers can be broadly classified into two different classes. There are different approaches to speech synthesis such as rule-based, articulatory modeling and concatenative technique. The best known method for speech synthesis is the articulatory method [2]. In this method the human larynx,the main speech production system is modeled electronically. However, recent speech research has been directed towards concatenative speech synthesizers due to difficulties in modeling the human larynx accurately. There are three basic approaches to concatenative synthesis. Unit selection synthesis uses a large database of segmented speech waveforms. At runtime, the best chain of candidate segments is selected from the database to form the desired output utterance. The size of the database is directly proportional to the quality of the output speech. Some examples of unit selection based synthesis are *Laureate* [3], *CHATR* [4], *Shruti* [1] etc.

A. Shruti: The Indian Language TTS System

A schematic diagram of the speech synthesis system selected for this work is shown in Fig.1 . The system consists of two main blocks: block A, the language dependent block and block B, the Indian Language Phonetic Synthe-

1-4244-0630-7/07/$20.00 ©2007 IEEE.

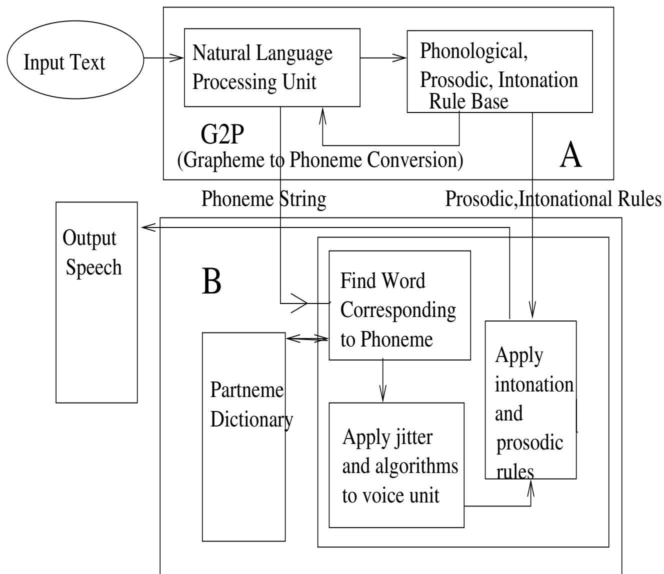

Fig. 1. Architecture of Shruti

sizer (ILPS). Block A consists of an input device, a natural language processor and an intonational and prosodic rule base. The natural language processor in block A comprises of a phoneme parser, which uses either a phonological dictionary, syllable and word marker to produce an output phonemic string. Part B is the synthesizer. The output speech is produced by taking the output string of phonemes (in ILPS symbols) and information for intonation and prosody from the basic rule base as input. Thus the output phonemic string is utilized by block B to produce the speech output. For this case study *Shruti* was cross-compiled for arm-linux platform and ported to OMAP1510 [5] which is a dual core architecture combining a TMS320C55xDSP core with a TI-enhanced ARM925T processor.

III. Related Works

An hardware architecture for TTS systems have been reported in [6] which proposes an ASIP solution for Parametric Speech Synthesis. Design of a speech synthesis ASIC, based on the line spectrum pair (LSP) scheme can be found in [7]. However, these works do not address concatenative speech synthesizers and they do not present any generic performance evaluation and architecture design based on standard embedded processors. Design of a speech ASIP for concatenative speech synthesizer can be found in [8] which considers an instruction set extension approach for performance improvement.

IV. Performance Analysis

The architecture exploration task in the present work begins with a workload analysis of the speech synthesis engine in the *Simplescalar* architecture simulator [9]. The baseline configuration was selected to model the StrongARM architecture.

A. Execution Profiles

The procedure level profile information were extracted for Shruti in ARM platform. The TTS engine has two main modules: *Analyze*, the Natural Language Processing (NLP) intensive language dependent block and *Generate*, the DSP intensive synthesizer. The most frequently used procedures in *Shruti* are *FindLength*,*verb_morph_analysis*,*Concatenate_Consonant_Vowel* and *filter* which are all components of the DSP phase of the speech synthesizer. In particular, the *filter* procedure, which does 4-point moving average filtering on audio signals, consumes 66.44% of the execution time for an input dataset of 600 characters as shown in Fig.2 .

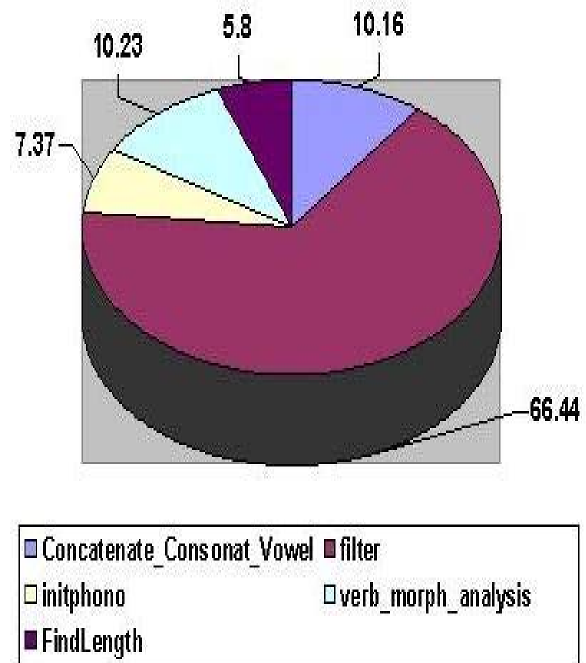

Fig. 2. Profile Results for Shruti

The execution time of the DSP phase increases with input text size as shown in Fig.3 . As the figure suggests, the execution time of the DSP phase, increases with input size in a sub-linear fashion and saturates to about 80% of the total execution time. Without loss of generality, it can be stated that concatenative speech synthesis applications spend most of the execution time in the DSP intensive phonetic synthesis phase.

B. Memory Demands

A detailed cache regression analysis was performed with varying block sizes and number of sets to provide some intuition into spatial versus temporal relationships in memory access. In this work, the memory system is modeled with access latencies of 1 cycle for L1 caches, 6 cycles for L2 caches and 18 cycles for the DRAM. Since, L1 caches have minimum access latency, L1 misses have the most pronounced effect on the number of execution cycles. The block size of both L1 instruction and data cache was varied

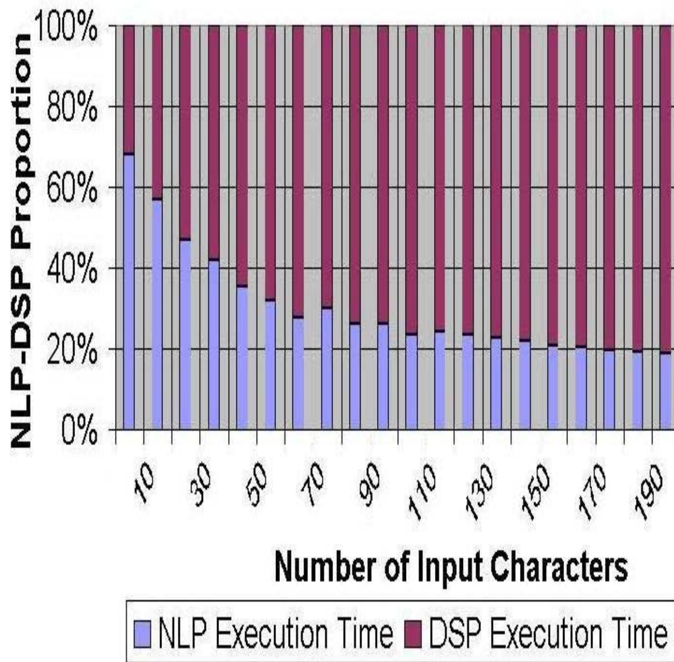

Fig. 3. Execution time of NLP and DSP tasks

from 8 to 64 bytes keeping the number of sets fixed at 128 followed by variation in number of sets upto 4096 keeping the block size fixed at 64. The reduction in miss% is evident from Fig.4 which shows the analysis for L1 instruction cache. Results are shown for three cache associativity values.

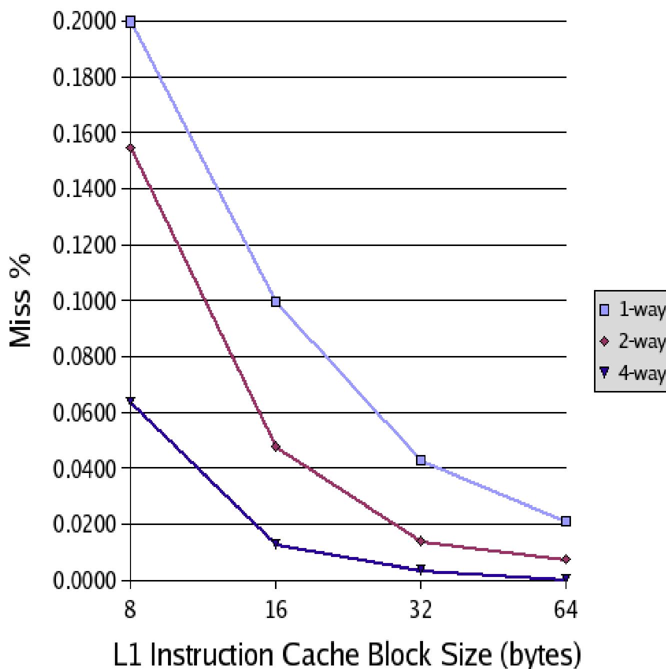

Fig. 4. Miss ratio for L1 instruction Cache with 128 sets

The block size variation leads to roughly 71% decrease in the number of execution cycles as shown in Fig.5 and the set size variation provides about 11% decrease in the cycle count.

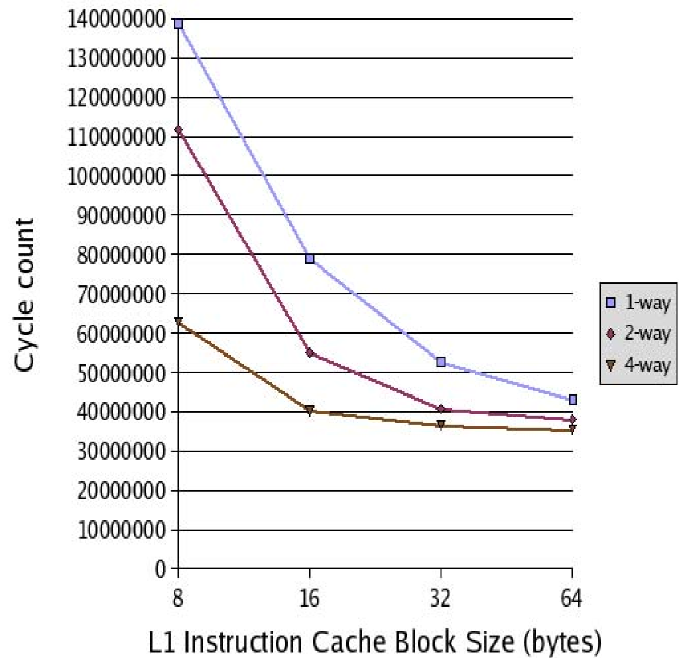

Fig. 5. Speedup due to Cache block-size variation

Performance evaluation was also performed assuming a bi-modal branch predictor hardware with branch history table size variation from 8 to 4096 which provided roughly 16% savings in execution cycles.

V. Architectural Revisions

The architectural revisions reported in this work are based on the performance analysis carried out assuming a baseline machine configuration. The analysis phase resulted in identifying the application bottlenecks and fixing up appropriate memory parameters for the TTS. This Section reports the gradual architecture refinements and their evaluations leading to the final architecture and its validation.

A. Exploiting Parallelism

The profile information reveals that the execution time of the DSP intensive module saturates to almost 3 times the execution time of the NLP module for high values of input string length. Hence, for achieving throughput values higher than what is possible by software only execution, one possible solution is to implement a multi-threaded architecture that accelerates the execution of the time-consuming moving-average filter computation loop in the DSP block by concurrent execution.

Performance results were obtained by porting the TTS into a multi-threaded version of the simplescalar simulator [10] with proper *store* and *allocate* locks implemented for data-points shared between the threads. The simulated

super threaded architecture has unified level-2 cache and symmetric threading units (TUs). The TUs were individually modeled on the simplescalar base processor platform with communication units attached in a ring topology and memory buffers. The architecture is modeled on the philosophy of *thread-pipelining* which allows threads with data and control dependencies to be executed in parallel and run-time checking [11]. The improvement in execution time of the phonetic synthesis phase due to the parallelization of the TTS code is evident from Fig.6 which also shows the impact of loop-unrolling and TU variation on the execution time of the parallel threads. The speedup obtained by loop-unrolling is noticeable specially in the case where the parallelization effort with 2 TUs and no unrolling lead to performance degradation due to the threading overhead.

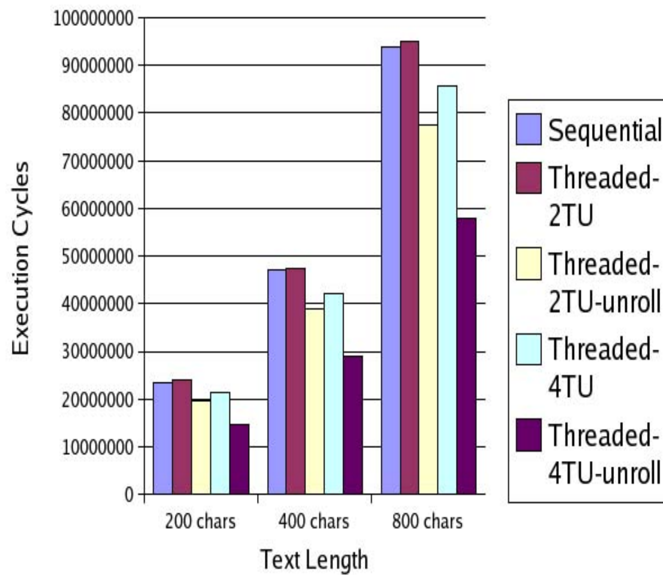

Fig. 6. Execution Pattern for 2 and 4 TUs

Although the multi-threaded architecture provided substantial speedup results for the TTS, implementing such a full-fledged high performance multi-processing environment seems to be an overkill for the concerned application keeping in mind the domain of low-power portable devices where such sophisticated memory systems, memory buffers and thread-level communication units of a high-performance superscalar core are not practically realizable. A learning from this initial revision work has been that the computation intensive DSP phase contains massive scope of parallelization which can be exploited by employing multiple low-power embedded cores like ARM and the performance penalty can be nullified by executing bottleneck code in custom hardware.

B. Accelerating the Bottlenecks

As reported previously, the 4-point moving average filter operation was found to be the bottleneck in the TTS system, consuming almost 70% of the TTS engine execution time for large input text length. This time-consuming operation can be accelerated using an application specfic custom accelerator. Although such an implementation will perform the operation in a single cycle, the communication overhead from the processor memory to the loosely coupled accelerator using external bus will lead to a massive overhead, thus finally killing the achieved speedup. Hence, the co-processor was tightly coupled with the processor core using a fixed sized shared memory. The software code was modified to prepare 100 data points from the initially synthesized and unfiltered speech waveform file in each iteration and map the values into the shared memory location to be processed by the co-processor. The processor will prepare the next set of data points in an overlapped manner. Signals for memory access control and synchronization, co-processor activation,reset,acknowledgement and halt were also implemented as part of establishing the communication between main processor and the co-processor. Due to limitation of instruction length, a custom instruction approach to solve the problem could take only two data points and perform single cycle add and shift thus requiring 3 instruction cycles in total for a 4-point moving average calculation. Such was not the case with the co-processor approach. In this case, add and shift of all four operands became a single cycle hardware operation. Due to the choice of a fixed window size, the shared memory implementation was possible thus helping to minimize the communication overhead. The resultant speed-up was about 2.1 times over the software only execution of the TTS. The result is shown for different input lengths in Fig.7 . Some statistics regarding the implemented co-processor is given in Table I. The *GeZel* [12]design language and environment was chosen for implementing and integrating the co-processor with *SimIt-Arm* [13], an instruction set simulator (ISS) of the StrongARM instruction set architecture (ISA).

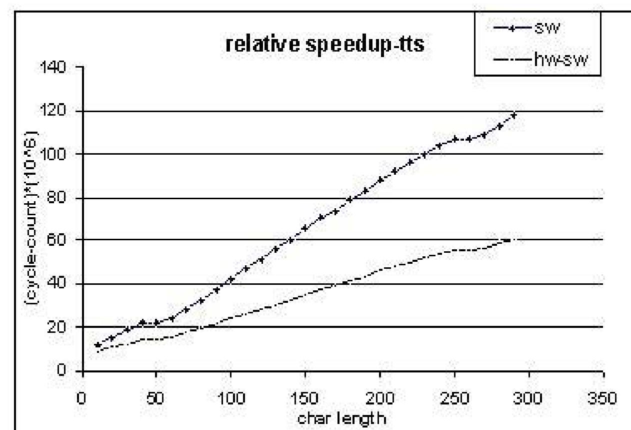

Fig. 7. TTS-speedup results

C. Final Architecture

From the procedure level analysis, it has been clear that the TTS can be modeled as a streaming application with

TABLE I
CO-PROCESSOR STATISTICS

Application	TTS
Operation	4point moving average
Computation Cycles	1
Lines of Gezel Code	271
Lines of VHDL generated	838
No. of Logic cells(Altera Stratix FPGA)	800
Area consumption(0.18μ process ASIC)	0.038 mm^2

consecutive tasks, each operating on the output data produced by its predecessor. The single-core based architecture proposed in Section V-B provides more than twice the throughput of the software-only execution. However, in order to meet throughput requirements higher than this while respecting the processor clock-speed, task-level partitioning into multiple PEs is required. Initially, the NLP and DSP tasks were partitioned into subsequent PEs for gaining the advantage of overlapped execution. The filter-accelerator unit is coupled with the PE which processes the DSP intensive speech-synthesis task. However, the execution pattern reveals that even with the accelerator unit, the execution time of the DSP task saturates to almost 2.5 times that of the NLP task with variation of the input text length (initially without the accelerator it saturated to about 4 times). Performance analysis of the DSP task revealed the huge scope of parallelization as shown in Section V-A. This immediately suggests the use of Symmetric Multiprocessing (SMP) or Simultaneous Multi-threading (SMT) style of architectures. However, the unsophisticated memory systems found in portable devices will lead to inefficient processor utilization for a standard arrangement of SMPs. On the other hand, a single SMT processor will require a massive number of resources for exploiting the available concurrency. Hence, keeping in mind the target domain of embedded handheld devices, this work proposes a hybrid architecture comprising of simple low-power processing elements based on the StrongARM ISA with an initial static partitioning of the speech synthesis tasks to maximize load balancing.

The proposed hybrid architecture exploits the available parallelism in the DSP phase by applying task-level threading techniques. Based on the ratio of execution times of the two coarse-grained tasks, it was initially assumed that partitioning the DSP task into three parallel PEs (an architecture comprising four PEs including the front-end NLP unit) will nullify the execution time gap. It was experimentally found that if the number of DSP threads is more than three, it leads to performance degradation due to the threading overhead and PE under-utilization because of the simple fact that the NLP unit was now taking more time to generate the phoneme tokens required by the DSPs. This will again call for task threading for the NLP unit and subsequent threading of the DSP units which takes the total number of cores from four to eight which is unrealistic keeping in mind the commercially available embedded multi-cores. With technology scaling and more number of cores in a low-power embedded processor in the future, the only thing required will be to continue threading of the pipelined tasks maintaining the proper ratio. Hence, the final architecture has been designed with 4 processing elements as shown in Fig.8 .

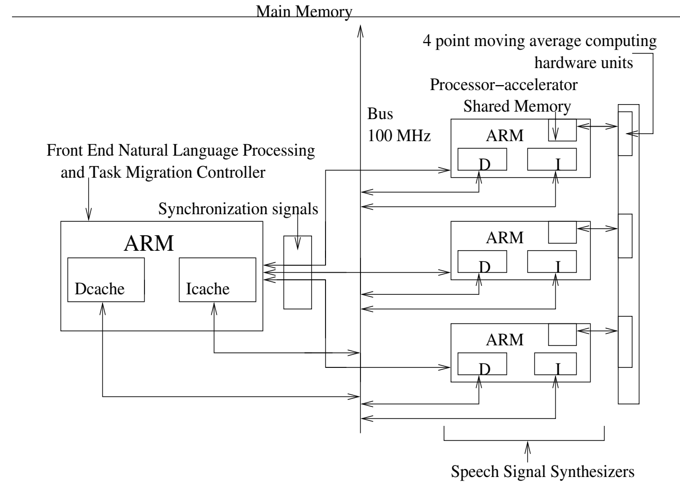

Fig. 8. Final Architecture

The proposed architecture implements a shared memory multi-processor system with four ARM processors. The StrongARM processor cores run at a frequency of 206 MHz. Each processor core has got its own data and instruction cache. No cache coherence protocol or hardware test-set locks have been implemented as data consistency has been maintained at the software level by partitioning and locking. The front-end NLP unit was also used as the controller for activating the PEs which performed the DSP task and synchronizing their access to the intermediate phonemes generated by natural language analysis and *grapheme-to-phoneme* conversion [14]. The synchronization signals from both sides were established by implementing a separate 4-bit bi-directional bus. Each of the DSP units were customized with closely coupled accelerators for moving average computation and shared memory based communication as described in Section B. The shared memory multi-processor architecture, accelerator units and communication infrastructure were developed using *GeZel* and *SimIt-Arm*.

An important point in this regard is that the throughput achieved by this architecture depends on the number of overlapped executions (the number of times this pipelined design operates) which again will depend on the number of input characters (a dynamic value for streaming text) that needs to be processed and the *processing-length* in each pass.

Performance Results: The throughput (in bytes/sec(bps)) offered by three architectural configurations for an input of 2000 characters is given in Table II along with area estimates.

For higher input values, the speedup is bound to increase as the pipeline will operate more number of times. In the final architecture, the front-end controller selects an execution mode based on the throughput requirement. For low throughput (th) requirement ($th \leq 500bps$), the

TABLE II
THROUGHPUT VS AREA

Architecture	Throughput(bps)	area mm^2
single ARM	515	11.8
ARM + accelerator	1081	11.838
4 ARM + 3 accelerators[a]	1700	47.314

[a]Final pipelined architecture working with *processing-length* of 150 chars creating $\lceil 2000 \div 150 \rceil = 14$ rounds of overlapped executions

controller itself performs both of NLP and DSP tasks. For $500bps \leq th \leq 1100bps$, the NLP task is executed in the controller and the DSP task is executed in one of the three PEs with accelerator. For higher throughputs, all the threading elements are activated. In each of the execution modes, the unused PEs are put into *idle* mode by the controller for power savings. The area and power consumption estimates of the cores at different power modes are taken from StrongARM data sheets [15]. Similar estimates for the accelerator are generated using Synopsis Design Compiler [16] with a 0.18μ process which is the same process technology used for the current ARM cores. Variation of average power consumption with throughput requirement of the final architecture is given in Table III.

TABLE III
AVERAGE POWER CONSUMPTION OF FINAL ARCHITECTURE

Throughput(bps)	Active units	Power(mW)
$th \leq 500$	Controller	700
$500 \leq th \leq 1100$	Controller, 1 PE, accelerator	1014.18
$th \geq 1100$	Controller, 3 PEs, 3accelerators	1642.55

VI. Conclusion

The present work considered domain specific architecture exploration for text to speech synthesis in embedded handheld devices. The design approach has largely been dominated by exploiting the available concurrency in the DSP intensive phase of speech synthesis and custom hardware implementation for application bottlenecks. With multiple revisions, a multi-processor architecture was arrived at which is flexible in terms of resource utilization depending on the performance requirement. The final architecture provided 3.3 times speedup over single processor execution for a test input of 2000 characters and as the trend shows, the speedup achieved increases with input size.

The architecture design work has been largely based on the rough assumption that the use of low-power embedded cores and simple custom hardware components will not create an overall system with prohibitive power consumption. Though this is a valid assumption in embedded system design (for example, cache power result is based on average dissipation value), a more detail power analysis by attaching cycle accurate power models of each of the individual cores, memory and cache components is necessary for an end-to-end design. The intention has been to achieve power-awareness by task migration and resource utilization depending on the performance requirement. However, finer control can be achieved by modifying the simulation infrastructure with frequency scaling of the PEs by the main controller (StrongARM can operate at 206 and 133 MHz). These architectural refinements are part of the intended future work.

References

[1] A. Mukhopadhyay, Monojit Chowdhury, Soumen Chakraborty, Soumyajit Dey, Anirban Lahiri and Anupam Basu, "Shruti - An Embedded Text to Speech System for Indian Languages," *IEE Proceedings on Software Engineering*, 153 (2): 75-79, April 2006.

[2] Bavegard, M., "Towards an articulatory speech synthesizer: Model development and simulations," *TMH-QPSR*, 1996.

[3] J. H. PAGE and A. P. BREEN, "The Laureate text-to-speech system - architecture and applications," *BT Technology Journal*, pp. 57-67, January 1996.

[4] A. Black and P. Taylor, "A. W. Black and P. Taylor. CHATR: a generic speech synthesis system," *COLING94*, Kyoto, Japan, 1994.

[5] OMAP1510. http://www.ti.com/.

[6] Ravi Saini, Pramod Tanwar, A. S. Mandal, S. C. Bose, Raj Singh and Chandra Shekhar, "Design of an Application Specific Instruction Set Processor for Parametric Speech Synthesis," *17th International Conference on VLSI Design*, pp. 773, 2004.

[7] Xingjun Wu and Yihe Sun, "LSP speech synthesis ASIC architecture," *Solid-State and Integrated Circuit Technology, 1995 4th International Conference*, pp. 700-702, 1995.

[8] Soumyajit Dey, Susmit Biswas, Arijit Mukhopadhyay and Anupam Basu, "An Approach to Architectural Enhancement for Embedded Speech Applications," *19th International Conference on VLSI Design*, 2006.

[9] Simplescalar. http://simplescalar.com/.

[10] Simca. http://www.arctic.umn.edu/SIMCA/index.shtml/.

[11] Jian Huang, "The Simulator for Multithreaded Computer Architecture," *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 00-05*, June 2000.

[12] Gezel. http://www.ece.vt.edu/schaum/gezel/.

[13] SimIt-Arm. http://sourceforge.net/projects/simit-arm/.

[14] Choudhury M., "Rule-based Grapheme to Phoneme Mapping for Hindi Speech Synthesis," *90th Indian Science Congress of ISCA*, Bangalore, 2003.

[15] StronARM. http://www.intel.com/.

[16] Design Compiler. http://synopsys.com/.