ANALYSIS OF A PARALLEL LEXICAL-TREE-BASED SPEECH DECODER FOR MULTI-CORE PROCESSORS

Naveen Parihar
Dept. of Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS 39762
email: np1@ece.msstate.edu

Eric A. Hansen
Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
email: hansen@cse.msstate.edu

ABSTRACT

We present a systematic analysis of a lexical-tree based parallel search algorithm for multi-core desktop processors. We introduce an analytical model that predicts the speedup from parallelization after accounting for load imbalance among the cores. Various sources of overhead in the parallel search algorithm are described, benchmarked and analyzed. Besides load imbalance, these include the inherently serial steps of the parallel search algorithm and an increase in main memory access latency.

1. INTRODUCTION

With multi-core architectures becoming more prevalent in desktop computers, applications including speech decoding must be parallelized to exploit the resources of the multiple cores. In [1], we presented a novel approach to parallelizing a lexical-tree based LVCSR decoding algorithm for multi-core desktop processors. The approach distributes the search among the cores by dividing the lexical tree in a way that minimizes communication between cores. We presented experimental results for running the decoder on the two cores of an Intel Core 2 Duo processor.

In this paper, we present experimental results for running the decoder on the four cores of an Intel Core 2 Quad processor and analyze the results in order to better understand the performance of the parallel search algorithm. We identify and benchmark the various sources of overhead in the parallel search algorithm. We also propose an analytical model that predicts the speedup after accounting for the overhead due to load imbalance. Although our experimental results are for a specific decoder and hardware platform, the analytical model we develop is general and a similar analysis can be applied to any other speech recognition decoder.

2. SYSTEM SETUP

The system setup used for our experiments consists of a speech recognition decoder, corpus, and computing platform.

- **Decoder:** The details of the serial and parallel search algorithms of the Mississippi State decoder are provided in [1, 2]. It is a hierarchical dynamic programming based time-synchronous Viterbi search engine that supports N-gram context dependent cross-word triphone decoding. It has been used on several evaluations including Aurora Evaluations [3].
- **Corpus:** Our parallel search algorithm was benchmarked on the 5K-word closed-loop WSJ0 task [4]. Phonetic decision-tree based state-tied cross-word speaker-independent triphone acoustic models with 16 Gaussian mixtures per state were trained using the SI-84 training set [5]. The Nov'92 NIST evaluation set (330 utterances), when decoded with a standard 5K lexicon and bigram language model, achieves a WER of 8.4%. Eight utterances were randomly selected from the evaluation set to facilitate rapid experimentation. All the timing measurements are averaged over five runs.
- **Hardware, OS and Compiler:** Experiments were run on an Intel Core 2 Quad (Model number Q6600) running Fedora Core 6 Linux operating system. Each of the four cores runs at a clock speed of 2.40 GHz. Each core has its own 32 KB L1 data cache and 32 KB L1 instruction cache. Each pair of cores shares one of the two available 4 MB L2 caches. The total size of L2 cache is 8 MB. The size of main memory is 2 GB. Note that the entire search space fits in main memory, and therefore page faults are not investigated in this research. The compiler used is GNU gcc version 4.1.1. The parallel search algorithm is implemented using the P-thread library. Assuming one thread per core, the terms thread and core are used interchangeably.

3. ANALYTICAL MODEL FOR LOAD IMBALANCE

Among several sources of overhead due to parallelization is load imbalance among the processor cores. We first develop an analytical model that captures the influence of load imbalance on parallel speedup.

The parallel search algorithm we consider is the Algorithm 2,3 presented in [1]. In our approach, the lexical tree is statically divided among the cores at the root node. The active search space during the search process then gets distributed among the cores based on the division of the lexical tree. The load imbalance occurs due to the uneven distribution of the active search space.

The parallel algorithm consists of both parallel steps and inherently serial steps. A detailed description of these steps is not provided here due to space limitations. In order to achieve good speedup, the time taken by each of the parallel steps during the processing of each frame should ideally be perfectly balanced between the CPU cores. We can estimate the time taken by each of these parallel steps based on one or more statistics about the search space. For example, the number of words that need to be expanded is a search space statistic that influences the time taken by the step *expand_words_to_states*. For each step of the algorithm, we identify a search space statistic that influences runtime the most. In case more than one search space statistic influences runtime, we empirically select the one that contributes most to the search step's runtime.

In general, we can predict the speedup for each parallel step, p, based on its search space statistic, s, as follows:

$$\hat{S_p} = \frac{serial\_runtime}{parallel\_elapsed\_runtime} = \frac{N_c \cdot \beta_s}{\beta_s + \alpha_s}, \quad (1)$$

where β_s is the perfectly balanced work due to search space statistic s, α_s is the load imbalance in search space statistic s, and N_c is the total number of cores.

The perfectly balanced work due to search space statistic s is given by:

$$\beta_s = \sum_{f=1}^{N_f} \beta_{s,f}, \quad (2)$$

where N_f is the total number of frames, and

$$\beta_{s,f} = \frac{\sum_{c=1}^{N_c} n_{s,f,c}}{N_c}, \quad (3)$$

where $n_{s,f,c}$ is the value of search space statistic s in core c during the processing of frame f.

The load imbalance in search space statistic s is:

$$\alpha_s = \sum_{f=1}^{N_f} \max_{c=1}^{N_c} \{n_{s,f,c} - \beta_{s,f}\}, \quad (4)$$

Note that for a perfectly balanced load, $n_{s,f,c} = \beta_{s,f}, for\ all\ c,f$. Hence, $\alpha_s = 0$, and $\hat{S_p} = N_c$.

The predicted speedup and the corresponding search space statistic used to estimate this speedup for all parallel steps is shown in Table 1. We observe that for 2-core, most of the parallel steps are reasonably balanced. However, for 4-core, there is greater load imbalance. Because imbalance in any step only influences the time spent in that specific step, we can apply these speedups to time taken by corresponding steps in serial search (Table 2), and add them to get a prediction of the elapsed runtime for these parallel steps. Adding the time taken by serial steps to the predicted runtime taken by parallel steps, provides the prediction of the elapsed runtime as shown below:

$$\hat{t} = \sum_{p=1}^{N_p} \{ts_p \cdot \hat{S_p}\} + \sum_{r=1}^{N_r} ts_r. \quad (5)$$

where N_p is the number of steps in serial search that can be parallelized, N_r is the number of inherently serial steps, and ts_x is the time taken by search step x.

We can then predict the speedup of the parallel search algorithm. The speedup prediction for 2-core is 1.52, and for 4-core is 2.29. The accuracy of our model might be improved by using more than one search space statistic to predict speedup for a search step. This possibility is left for future work.

4. OVERHEAD IN PARALLEL SEARCH ALGORITHM

In this section, we analyze and discuss various sources of overhead in the parallel search algorithm.

parallel step	search space stat	2cores	4cores
exp_wd_paths	#Word-ends	1.97	3.85
exp_phn_paths	#Phone-end Paths	1.65	2.31
prune_model_inst	#Mod. Inst. Pruned	1.84	2.92
project_states	#State Paths	1.87	3.25
prune_state_paths	#State Paths Pruned	1.84	2.98
likelihood_comp	#States Evaluated	1.80	2.68
LM_lookups	#Lookups	1.63	2.33
LM_lookaheads	#Lookaheads	1.79	2.87

Table 1. Predicted speedup ($\hat{S_p}$) considering load imbalance.

		%time	time(s)	xRT
Search Mang.	*exp_wd_paths_to_states*	30.73	203.44	3.72
	exp_phn_paths_to_states	9.71	64.26	1.18
	compute_max_phone model_inst_thresh	**8.65**	**57.25**	**1.05**
	prune_phn_model_inst	11.59	76.70	1.40
	project_states	19.05	126.12	2.31
	prune_state_paths	10.82	71.65	1.31
	create_word_paths	**1.46**	**9.66**	**0.18**
	compute_word-end_ pruning_thresh	**0.01**	**0.08**	**0.01**
	initialization	**0.54**	**3.56**	**0.07**
	total	92.55	612.72	11.20
state_likelihood_computation		4.34	28.70	0.53
LM comp.	*LM_lookups*	2.66	17.63	0.32
	LM_lookaheads	0.43	2.84	0.05
	total	3.3	20.47	0.4
total		100.0	661.89	12.11
total serial		**10.65**	**70.55**	**1.29**
total parallel potential		89.35	591.34	10.82

Table 2. Serial search algorithm's timing measurements (ts_x). Inherently serial steps are shown in bold case.

4.1 Overhead in Parallel Steps

Let ts_p denote the time spent by the serial search algorithm to process a step p. Let t_p denote the time spent by the parallel search algorithm to process the parallel step p using N_c cores. The overhead for parallel step p is defined as follows, by the standard parallel computing analysis [6]:

$$top_p = N_c \cdot t_p - ts_p, \quad (6)$$

The total parallel cost, $N_c \cdot t_p$, is the sum of cost for all cores. If N_f frames are processed in parallel step p, the cost per core for processing each frame is the sum of the time spent executing the parallel step p and idling due to load imbalance, for the core. The total parallel cost is defined as:

$$N_c \cdot t_p = \sum_{c=1}^{N_c} \sum_{f=1}^{N_f} \{tp_{p,f,c} + ti_{p,f,c}\}, \quad (7)$$

where $tp_{p,f,c}$ is the time taken by core c to process parallel step p during frame f, and $ti_{p,f,c}$ is the idling time due to load imbalance. Rearranging Equation (7) and substituting in Equation (6):

$$top_p = \{\sum_{c=1}^{N_c} \sum_{f=1}^{N_f} tp_{p,f,c} - ts_p\} + \sum_{c=1}^{N_c} \sum_{f=1}^{N_f} ti_{p,f,c}. \quad (8)$$

We define the first term as *overhead due to parallel execution*, $topp_p$, and the second term as *overhead due to load imbalance*, $topi_p$.

$$topp_p = \sum_{c=1}^{N_c} \sum_{f=1}^{N_f} tp_{p,f,c} - ts_p, \qquad (9)$$

$$topi_p = \sum_{c=1}^{N_c} \sum_{f=1}^{N_f} ti_{p,f,c}, \text{ and} \qquad (10)$$

$$top_p = topp_p + topi_p. \qquad (11)$$

Table 3 compares the actual overhead due to load imbalance ($topi$), and predicted overhead due to load imbalance ($\widehat{topi}$) using the analytical model introduced in Section 3.

$$\widehat{topi}_p = N_c \cdot \frac{ts_p}{\hat{S}_p} - ts_p = ts_p \cdot \{\frac{N_c}{\hat{S}_p} - 1\}. \qquad (12)$$

Note that the predicted overhead is an overestimate of the actual overhead. As described in Section 3, the accuracy of the model can be improved by incorporating more than one search space statistic for certain search steps, if required. Nonetheless, the model based on one search space statistic per search step is useful for feasibility analysis. Both the predicted and actual load imbalance show that the load imbalance in 4-core parallel search is approximately 3.5 times the load imbalance on 2-core parallel search.

4.1.1 Overhead Due to Parallel Execution

Three sources of *overhead due to parallel execution* ($topp$) are:

- *Excess Computations due to an increase in likelihood computations*: This overhead is incurred only by the likelihood computation step. Each thread keeps its own local copy of the likelihood cache and hence, the states that fall in multiple threads get evaluated multiple times. Table 4 presents the increase in number of likelihood computations. Assuming equal cost for each likelihood computation during the serial search, we can estimate the time taken by likelihood computations in the parallel search, also shown in Table 4.
- *Excess Computations due to changes required to convert serial code into parallel code*: Changes in the algorithm might increase work. Also, the compiler might produce a different set of processor instructions due to the changes in the code.

parallel step	2 cores		4 cores	
	$\widehat{topi}_p$	$topi_p$	$\widehat{topi}_p$	$topi_p$
exp_wd_paths_to_states + *exp_phn_paths_to_states* + *LM_Lkups*+*LM_Lkah.*	21.06	15.83	68.69	50.06
prune_phn_model_inst	6.66	4.92	28.36	17.49
project_states + *state_likelihood_comp*	11.95	14.48	43.24	55.30
prune_state_paths	6.23	4.03	24.52	14.06
total	45.90	39.26	164.81	136.91

Table 3. Comparison of actual ($topi_p$) and predicted overhead ($\widehat{topi}_p$) due to load imbalance in seconds.

	1 core	2 cores		4 cores	
#Comp.	1413.8	1793.6	(+379.8)	2054.5	(+640.7)
Time	28.70	36.40	(+7.70)	41.70	(+13.00)

Table 4. Average number of likelihood computations per frame, and corresponding estimated parallel runtime in secs.

- *Increase in CPU Pipeline Stalls due to hardware limits*: For example, on an Intel Core 2 Quad processor, parallel concurrent execution increases the front side bus-bandwidth utilization and main memory access latency.

Table 5 presents a comparison of $topp_p$ for all the parallel steps among serial (1-core), 2-core parallel, and 4-core parallel search. Due to space limitation, instead of considering the behavior of each parallel step individually, we analyze the overall behavior. The overhead for 2-core search is 77.12 seconds. The overhead for 4-core search increases by approximately three times, and is 267.89 secs. The contribution from *Excess Computations due to an increase likelihood computations* to the total overhead is very small (7.7 secs for 2-core and 13.0 secs for 4-core) and can be ignored.

The total overhead ($topp$) due to parallel execution can then be decomposed into the two remaining sources by observing the measurements of processor-specific high performance events shown in Table 6. *Excess Computations due to changes required to convert serial code into parallel code* result in time overhead for an increase in non-stalled CPU cycles (12.28 secs for 2-core and 26.00 secs for 4-core). The overhead due to *Increase in CPU Pipeline Stalls* dominates the total overhead. For 4-core parallel search, this overhead is 241.86 secs which approximately 3.7 times the overhead for 2-core parallel search (64.84 secs). In the next section, we present a detailed analysis of the factors that contribute to the *Increase in CPU Pipeline Stalls*.

4.1.2 CPU Pipeline Stalls

Table 6 presents the decomposition of time due to CPU pipeline stalls among four major performance events. These events include stalls due to Branch-MisPrediction, Non-prefetched Retired Load L2 Cache Hits, Non-prefetched Retired Load L2 Cache Misses, and Non-prefetched Retired Load Data-TLB Cache Misses. The number of cycles stalled due to Branch-MisPrediction can be directly measured; from this, the corresponding time is computed. For the other three events, the penalty in the form of number of stalled cycles needs to be estimated and this can result in an overestimation. An estimate of 10 cycles for Data-TLB cache miss penalty

parallel step	2 cores	4 cores
expand_word_paths_to_states	-5.2	23.42
expand_phone_paths_to_states	1.45	4.91
prune_phone_model_instances	25.36	73.14
project_states	31.84	66.35
prune_state_paths	20.12	68.33
state_likelihood_computation	11.82	27.20
LM_lookups	0.08	2.42
LM_lookaheads	1.58	2.09
total	87.05	267.89

Table 5. Overhead due to parallel execution ($topp_p$) in secs.

	1 core	2 cores		4 cores	
# Retired Load L2 Hit	1,759,406,790	1,952,197,334	(+10.95%)	2,202,346,210	(+25.17%)
# Retired Load L2 Miss	4,356,423,142	4,355,233,453	(-0.02%)	4,043,894,647	(-7.17%)
# Retired Load Data-TLB Miss	4,372,081,635	4,110,294,090	(-5.98%)	3,693,061,200	(-15.53%)
Total # Outstanding Load Bus Reqs./cyc	3,034,881,101,734	3,217,675,629,573	(+6.02%)	4,256,667,328,050	(+40.25%)
Total # Load Bus Request	12,849,414,742	11,812,248,446	(-8.07%)	10,544,482,341	(-17.93%)
Access Latency in cycles	236.18	272.40	(+15.33%)	403.68	(+70.91%)
Total Time (s)	591.34	678.39	(+87.05)	859.20	(+267.86)
Time taken by Non-stall Cycles (s)	163.28	175.56	(+12.28)	189.28	(+26.00)
Time taken by Pipeline Stall Cycles (s)	428.06	492.90	(+64.84)	669.92	(+241.86)
Stalls due to Branch-MisPrediction (s)	5.95	9.72	(+3.77)	16.27	(+10.32)
Stalls due to Retired Load L2 Hit (s)	10.26	11.38	(+1.12)	12.84	(+2.58)
Stalls due to Retired Load L2 Miss (s)	428.67	494.27	(+65.60)	680.12	(+251.45)
Stalls due to Data-TLB-Load-Miss (s)	18.21	17.12	(-1.09)	15.38	(-2.83)

Table 6. Important performance measurements for parallel steps.

for the Intel Core 2 architecture was taken from [7]. An L2 cache hit penalty of 14 cycles was estimated using [8]. The L2 cache miss penalty is computed as main memory access latency, as described in [9]. As shown in Table 6, the access latency varies for serial (1-core), 2-core parallel, and 4-core parallel search. This leads to the following important observation: concurrent execution among multiple cores results in a greater number of load bus requests per cycle. This happens due to a larger number of L2 cache misses generated as a result of the load requests from multiple cores in a given time interval as compared to serial search. Hence, the main memory access latency increases.

The estimated time due to stalls caused by Non-prefetched Retired Load L2 Cache Misses dominates the overall time due to stalls. The number of Non-prefetched Retired Load L2 Cache Misses decreases in parallel search because of an increase in overall L1 data cache size (each core has its own L1 cache) and L2 cache size (for 4-core parallel search, both the available L2 data caches are used). However, the main memory access latency increases which results in an increase in stall time due to Non-prefetched Retired Load L2 Cache Misses. It is obvious that the estimated L2 cache miss penalty is an overestimate. Nonetheless, it provides useful information required to characterize the overhead. Efforts to reduce the overall stall time should involve optimizations that reduce the overall bus-traffic and Non-prefetched Retired Load L2 Cache Misses.

4.2 Overhead due to Serial Steps

When processing a serial step, r, of the parallel search algorithm, all but one of the cores is idling. This results in an overhead, tor_r, due to serial step r,

$$tor_r = \{N_c - 1\} \cdot \sum_{f=1}^{N_f} tr_{f,r}, \qquad (13)$$

where $tr_{f,r}$ is the time taken by a single core to process serial step r during frame f. In Section 4.4, it is shown that the overall time taken by the serial steps is comparable among serial (1-core), 2-core parallel, and 4-core parallel search. The overhead tor_r for all serial steps is shown in Table 7. As expected, the total overhead due to inherently serial steps in 4-core parallel search is approximately three times the overhead in 2-core parallel search.

serial step	2 cores	4 cores
compute_max_phone_model_inst_thresh	57.19	166.77
create_word_paths	10.88	36.72
lex-tree_synchronization	0.94	4.08
compute_word-end_pruning_thresh	0.36	1.68
initialization	3.85	10.17
read_featVector	0.05	0.48
total	73.27	219.90

Table 7. Overhead due to serial steps (tor_r) in seconds.

overhead	2 cores		4 cores	
$\sum_{p=1}^{N_p} topi_p$	39.26	19.71%	136.91	21.92%
$\sum_{p=1}^{N_p} topp_p$	87.05	43.60%	267.89	42.88%
$\sum_{r=1}^{N_r} tor_r$	73.27	36.69%	219.90	35.20%
to	199.68	100.00%	624.70	100.00%

Table 8. Distribution of total overhead in seconds.

4.3 Total Overhead

The total overhead, to, of the parallel search algorithm is given by:

$$to = N_c \cdot t - ts = \sum_{p=1}^{N_p} topp_p + \sum_{p=1}^{N_p} topi_p + \sum_{r=1}^{N_r} tor_r, \qquad (14)$$

where t is the total elapsed runtime of the parallel search algorithm, ts is the total runtime of the serial search algorithm, N_p is the total number of parallel steps, and N_r is the total number of serial steps. Table 8 shows the distribution of the total overhead. The overhead due to inherently serial steps (tor) is about 35%, and cannot be eliminated. Ideally, we want this overhead to be the only overhead in parallel search. Hence, future research efforts need to focus on reducing the overhead in parallel steps due to load imbalance ($topi$) and parallel execution ($topp$).

4.4 Elapsed Timing Analysis

The total elapsed runtime, t, of parallel search is given by:

$$t = \sum_{f=1}^{N_f} \{ \sum_{p=1}^{N_p} \{ \max_{c=1}^{N_c} t_{f,p,c} \} + \sum_{r=1}^{N_r} t_{f,r} \}, \qquad (15)$$

	1 core	2 cores		4 cores	
expand_word_paths_to_states + *expand_phone_paths_to_states* + *LM_lookups+LM_lookaheads*	288.17	150.29	(-137.88)	91.10	(-197.07)
compute_max_phone_model_inst_thresh	**57.25**	**57.19**	**(-0.06)**	**55.59**	**-(1.66)**
prune_phone_model_inst	76.70	53.44	(-23.26)	41.62	(-35.08)
project_states+state_likelihood_comp	154.82	106.40	(-48.42)	76.05	(-78.77)
prune_state_paths	71.65	47.81	(-23.84)	38.87	(+32.78)
create_word_paths	**9.66**	**10.88**	**(+1.22)**	**12.24**	**(+2.58)**
lex-tree_synchronization	**0.00**	**0.94**	**(+0.94)**	**1.36**	**(+1.36)**
compute_word-end_pruning_thresh	**0.08**	**0.36**	**(+0.28)**	**0.56**	**(+0.48)**
initialization	**3.56**	**3.85**	**(+0.29)**	**3.39**	**(-0.17)**
read_featVector	**0.00**	**0.05**	**(+0.05)**	**0.16**	**(+0.16)**
total	661.89	431.21	(-230.68)	320.94	(-340.95)
speedup	1.00	1.54		2.06	
total serial	**70.55**	**73.27**	**(+2.72)**	**73.30**	**(+2.75)**
total parallel	591.34	357.94	(-233.40)	247.64	(-343.70)

Table 9. Elapsed timing measurements in seconds. Serial steps are shown in bold letters.

Equation (15) can be rearranged as follows:

$$t = \sum_{p=1}^{N_p} \{ \sum_{f=1}^{N_f} \max_{c=1}^{N_c} t_{f,p,c} \} + \sum_{r=1}^{N_r} \sum_{f=1}^{N_f} tr_{f,s}. \qquad (16)$$

The first term $\sum_{f=1}^{N_f} \max_{c=1}^{N_c} t_{f,p,c}$ represents elapsed time for parallel step p. The max operator indicates that if there is any load imbalance among the cores, the elapsed time will be equal to the time taken by core with maximum load. Other sources of overhead discussed in Section 4.1 increase the elapsed time also. The second term in Equation (16) represents the serial steps and includes the overhead discussed in Section 4.2. The comparison of elapsed time for the parallel and serial steps among serial (1-core), 2-core parallel and 4-core parallel search is shown in Table 9. The total time due to serial steps is almost constant in serial (1-core), 2-core parallel and 4-core parallel search. The overall speedup for 2-core parallel search is 1.54, and for 4-core parallel search is 2.06.

5. CONCLUSIONS AND FUTURE WORK

This paper presents a detailed analysis of an approach to parallelizing a lexical-tree based search algorithm for LVCSR on a multicore architecture. We propose an analytical model that is useful for predicting speedup for feasibility analysis. Various sources of overhead in the parallel search algorithm are identified, described and analyzed. The overhead due to inherently serial steps cannot be eliminated. The two sources of overhead in the parallel steps that can potentially be reduced are load imbalance and increase in CPU pipeline stalls. The overhead due to load imbalance in 4-core parallel search is approximately 3.5 times the load imbalance in 2-core parallel search. The overhead due to increase in CPU pipeline stalls in 4-core parallel search is approximately 3.7 times the overhead in 2-core parallel search. The increase in main memory access latency dominates the increase in overhead due to increase in CPU pipeline stalls. In the future, we plan to explore more sophisticated load balancing techniques and ways to reduce main memory access latency that could result in improved scalability.

REFERENCES

[1] N. Parihar and E. Hansen, "A Lexical-tree Division-based Approach to paralleize a Cross-word Speech Decoder for Multi-core Procesors ," in *Proc. EUSIPCO 2008*, Lausanne, Switzerland, August 2008.

[2] N. Deshmukh et al., "Hierarchical Search for Large Vocabulary Conversational Speech Recognition," *IEEE Signal Processing Magazine*, vol. 16, no. 5, pp. 84-107, September 1999.

[3] N. Parihar et al., "An Analysis of the Aurora Large Vocabulary Evaluation," in *Proc. EUROSPEECH 2003*, Geneva, Switzerland, September 2003, pp. 337-340.

[4] D. Paul and J. Baker, "The Design of Wall Street Journal-based CSR corpus," in *Proc. ICSLP 1992*, Banff, Alberta, Canada, October 1992, pp. 899-902.

[5] N. Parihar, *Performance Ananlysis of Advanced Front Ends on the Aurora Large Vocabulary Evaluation*. M.S. thesis, Department of Electrical and Computer Engineering, Mississippi State University, USA, December 2003.

[6] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Second Edition, Pearson Education, 2004.

[7] D. Levinthal, Execution-based cycle accounting on Intel Core 2 Duo processors. http://www.devx.com/go-parallel/Link/33315

[8] L. McVoy and C. Staelin, Lmbench - Tools for Performance Analysis, 1998. http://www.bitmover.com/lmbench.

[9] Stéphane Eranian, "What can performance counters do for memory subsystem analysis?," in *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, Seattle, Washington, 2008.