Performance Analysis of Speech Recognition Software

Chunrong Lai, Shih-Lien Lu+ and Qingwei Zhao
China Research, Intel Labs.
(chunrong.lai@intel.com, qingwei.zhao@intel.com,
+Microprocessor Research, Intel Labs.
5350 NE Elam Young Parkway, Hillsboro, OR 97124
(shih-lien.l.lu@intel.com)

Abstract

This paper characterizes the behavior of a speaker-independent large vocabulary continuous speech recognition (LVCSR) system. This system is used to dictate Chinese (Mandarin) utterances of different speakers and achieves a word recognition accuracies of 85%~96% depending on the cleanness of input signals and the complexity of the spoken sentences. Several methods are used to characterize its processing behavior. First, the same system is run on different Intel platforms and their performance measured. Second, we use an Intel performance monitoring toolset – Vtune to read hardware counters build in the CPU. These counters measured the instruction distribution as well as processor utilization rate. Third, a full platform simulator SoftSDV together with a cache simulator is used to study its memory behavior in more detail. We find this software system to have a large memory working set. Data access to first level cache has good locality. There are two groups of memory usage displacing each other in the second level cache. Second level cache miss rate declines much fast after the size increases beyond 8MB. We also identify a few instructions that cause a larger number of level-2 cache misses. Using software prefetching we improve the overall performance by an average of 7%.

1. Introduction

Speech recognition (SR) by computer has been an active research for a while now. It provides a natural interface for human to interact with machines. Many speech recognition systems are available in the market such as Dragon Systems' Dragon Naturally Speaking [1], IBM's viaVoice [2], L&H's Voice Xpress [3] and Philips' FreeSpeech [4]. However, general adoption of machine speech recognition is still not widely accepted due to recognition unreliability. Much research efforts continue to be devoted in perfecting speech recognition techniques in recent years. So far the most widely accepted technique is based on hidden Markov Model (HMM). It is used in most state-of-the-art SR systems in the world. These large vocabulary continuous speech recognition systems based on HMM often face the issue of trading-off between the recognition accuracy and the computation speed. Thus it is important to understand the behavior of LVCSR in order to improve its performance.

Unfortunately processor performance characteristics of speech recognition applications are not well published. Agaram et. al. [5] analyzed the characteristics of a public domain speech recognition engine called Sphinx [6]. This engine has a vocabulary of 21000 and is based on semi-continuous hidden Markov Model (SCHMM). Intel Labs, Intel China Research Center (ICRC), has developed a LVCSR engine that was originally licensed from Oregon Graduate Institute (OGI) [7][19]. The vocabulary of this system is over 51000 words and is based on continuous hidden Markov Model (CHMM) with multivariate mixture Gaussian as observation density to cover speech signal variability. With sufficient training data, CHMM systems gives better recognition performance. However, these systems are more complex and slow down recognition speed. ICRC's LVCSR engine has been extensively tested with various Chinese (Mandarin) speakers and can achieve speaker-independent word recognition accuracies of 85%~96%. The variation of accuracy is due to the cleanness of input signals and the complexity of the spoken sentences. In this paper, we examine the behavior of ICRC's LVCSR speech-engine.

We study this engine with several tools. First, we measure the running time of this LVCSR engine on different Intel Architecture based platforms. These platforms vary in CPU frequency and level-2 cache size. We, then, use a performance analysis tool called VTune™ [8] to uncover more detail characteristics of the program. Vtune supports both event and time based sampling. It also allows users to perform call graph profiling. We mainly use the event-based sampling (EBS) capability to collect run-time information by reading the performance monitoring counters on the processor. During event-based sampling, VTune interrupts the processor after a specified number of event occurrences, and collects a sample containing the instruction addresses where the event occurs. At the end of EBS, we determine how many times an event occurred by the collected data. Information collected with VTune includes instruction distribution, branch misprediction rate, cache misses, and instruction/micro-op executed per cycle. These

data are compared with some of the SPEC 2000 CPU benchmark programs [9]. As expected, there are many floating-point computations in this program. However many of them are of the form ADD/SUB followed by Multiply instead of the other way around. Moreover, we observe that around 65% of the time the processor is not retiring any instructions/micro-ops while running this LVCSR program. It also has a relative large second level cache miss rate. We perform further study on the memory behavior of this program using software simulation.

In order to consider all the effects, including system calls, we decide to use a full system simulator. We ran this LVCSR program on this full system simulator called SoftSDV [10], in batch mode using speech input captured in files. Due to long simulation time we generate memory traces and run these traces through a trace-driven cache simulator with different cache configurations. The results show that this engine in particular, and speech recognition applications in general, has large memory working sets. Memory references have good spatial locality and not as good temporal locality. Level 2 (L2) cache's miss ratio changes at a different rate after its size grow larger than 8 MB.

2. General features of LVCSR

A speech recognition system converts speech into text strings. An uttered sentence is digitized first. These digitized samples are grouped in overlapping frames. A set of features [11] capturing the characteristics of a frame is extracted. This set of features is referred to as an observation. Thus, an uttered sentence is represented by a group of observations: $O = o_1, o_2, o_3, \ldots o_{t-1}, o_t$. The goal of the speech recognition system is to find the most possible word-sequence, $W = w_1, w_2, w_3, \ldots w_{n-1}, w_n$, that matches the observations. This process is express mathematically [12] as:

$$W = arg\ max_W\ P(W/O) \quad (1)$$

The right hand side of equation (1) can be re-written due to *Bayes* rule:

$$W = arg\ max_W\ P(W)P(W/O)/P(O) \quad (2)$$

In equation (2), the probability of observed input sequence *P(O)* is constant. The probability of a word sequence *P(W)* is part of the *language model* and acts as a search constraint. The conditional probability *P(W/O)* measures how well the word string *W* matches the given input *O*. This conditional probability is obtained during the training phase and is usually referred to as the *acoustic model*.

Acoustic models are built for pre-specified acoustic units. In our system, phonemes (phones) are the basic acoustic units. Each phone in the language is modeled by an HMM. An HMM consists of two stochastic processes. One is a *hidden* Markov chain, which accounts for *temporal* variability. The other is an observable process, which accounts for *spectral* variability [13]. These two have proven to be very powerful to cope with most sources of speech ambiguity, and very flexible to allow the realization of recognition systems with dictionaries of tens of thousands of words. Each HMM consists of several states and different phones may share a state. Several phones are grouped to form a word, and words are collected to construct a sentence. Therefore a given sentence can be thought of as a collection of phones. There are thousands of HMM states in our system. Figure 1 illustrates an example HMM, which contains three regular states and two pseudo states (a start and an end).

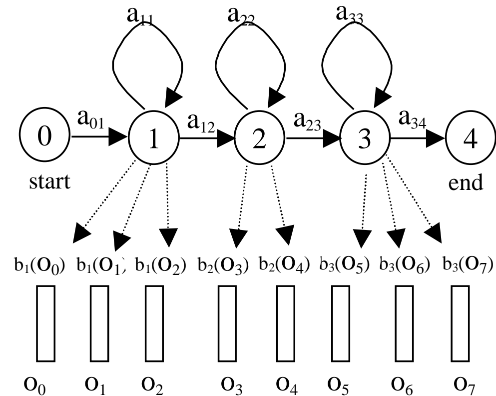


Figure 1. A five-state HMM with two pseudo states

The transition from state *i* to state *j* is determined by the transition probability a_{ij}. At each time unit, a speech observation o_t is matched according to the output density function $b_i(o_t)$. HMMs can be classified according to the nature of the elements of the *B* matrix, which are distribution functions. *Discrete HMMs' (DHMMs)* distributions are defined on finite spaces. In this case, observations are characterized as discrete symbols chosen from a finite alphabet . In detail, each incoming observation vector is replaced by the index of the closet vector in a precomputed codebook, and the output probability functions are just lookup table containing the possibilities of each possible VQ (vector quantization) index. Distributions are associated with model transitions. A transition probability and an output distribution on the symbol set are associated with every transition.

Another possibility is to define distributions as probability densities on continuous observation spaces. In this case, strong restrictions have to be imposed on the functional form of the distributions, in order to have a manageable number of statistical parameters to estimate. The most popular approach is to characterize the model transitions with mixtures of base densities function having a simple parametric form. The base densities function is usually Gaussian or Laplacian. The mean vector and the covariance

matrix can parameterize the base densities function . HMMs with these kinds of distributions are usually referred to as *continuous HMMs (CHMMs)*. In order to model complex distributions in this way a rather large number of base densities has to be used in every mixture. This may require a very large training corpus of data for the estimation of the distribution parameters.

In *semicontinuous HMMs (SCHMMs),* for example [14], all mixtures are expressed in terms of a common set of base densities. Different mixtures are characterized only by different weights. A common generalization of semicontinuous modeling consists of interpreting the input vector as composed of several components, each of which is associated with a different set of base distributions. The components are assumed to be statistically independent. The distributions associated with model transitions are products of the component density functions. Computation of probabilities with discrete models is faster than with continuous models, nevertheless it is possible to speed up the mixture densities computation by applying vector quantization (VQ) on the Gaussians of the mixtures [15]. Interested readers may find more details on their differences from [16].

As mentioned, in CHMM each HMM state contains large number of parameters of Gaussian distribution. In our system, each state requires 3 ~ 4KB of space to store these parameters. With thousands of state in our system the amount of storage required to store these parameters ranges in tens of million bytes (MB). Each time an input frame (represented as features) comes, Gaussian computation is processed using these parameters to get a probability of how close the observation (frame) matches a state is. Pronunciation dictionary is organized as a lexicon tree as illustrated by Figure 2 for searching. When the searching process reaches the leaf nodes of a lexicon tree, known as *word boundary*, the language models are accessed to get a score in addition to the acoustic score. Thus the recognition computation involves the calculation of the likelihood function $P(O|\lambda)$, where λ is a HMM. The recognition process consists of searching through all possible phones sequence, by looking up the acoustic model and the language model, and then finds the most meaningful (or with the highest score) sequence. In order to reduce the amount of possible branches needs to be searched, many unlikely paths are pruned during the search process.

The storage requirement of the language model is determined by the size of the vocabulary and is usually larger than the acoustic model. With a vocabulary of more than 51,000 words, the size of the language model and language look-ahead model is around 110MB. Besides the acoustic model and language model, there is an additional storage space requirement for the recognition process. During the search process, space is needed to construct lexicon tree and all current likely paths under consideration. This is named the search buffer and its size ranges in tens of MB also. When the search process arrives at the word boundary, accesses to the search buffer and the language model interact with each other. During the normal search mode, accesses to the search buffer and the acoustic model are interleaved. All these three spaces are large in size and may displace each other from cache memory.

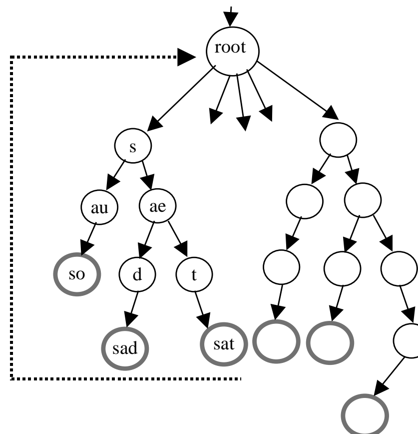


Figure 2. Lexicon tree

The technique named language model look-ahead is used [17][18] to reduce possible search space, thus reduces the computation time and search buffer space. This technique adds language model information into the node of a lexicon tree and further prunes the tree according to it. With this look-ahead technique fewer searches can reach the word boundary [19]. It increases the access frequency of the language model. However, it is still less than the access frequency of the acoustic model. The search buffer maintains dynamic data structures used to store all possible recognition results. These structures are very input dependent. For example, with noisy speech signals the size of search buffer may be tens of the size of clean speech signals.

During the search, beam values can be used to help decide if a search path should continue. If a score is not much lower than the current highest score according to the beam value, the path is likely to survive. We will keep the path in the search buffer. Thus the larger the beam value, the more space is needed to keep all temporary data around. More computations and search effort are required also. Matching (Gaussian computation) of continuous HMM to find the most likely phone spoken requires a lot of computation. Moreover, maintaining of the uncertain number of tokens in such a big tree also needs to be handled carefully. Good pruning algorithm can reduce the number of active nodes, active states and active tokens dramatically. For example, in our current system the active number of states is generally less than 1% of the total number. Many other optimization techniques at the implementation level have been applied to the speech

recognition system. For example, the pointer-base lexicon tree can be reordered based on some chunk with better data locality. Prefetch instructions can be employed inside the Gaussian mixture computation of the large HMM state since the data locality is better there. Also with the access locality, the language model look-ahead [7][18] probability of a node can be computed speculatively and buffered with the computing of its brother nodes. We have mentioned that different nodes may share a same trained state. A buffer is used for remembering the results of these Gaussian mixture computations. Thus, we avoid duplicating these Gaussian mixture computations if the probability has already existed in the buffer. Also because of the continuity of the voice, when a frame is computed with a HMM state, the following several frames can also be computed with the same state speculatively so as to utilize the state data more efficiently to avoid future stalls. This buffer becomes the fourth memory working set of the system. Now in the processing of normal nodes, the buffer interleaved with the search space, only when the buffer misses, the HMM states of the acoustic model are accessed. All these optimizations and speculative computations increase the data locality of the system and speed up the system. We also use tuned performance library [20][21] with Streaming SIMD Extension (SSE) instructions as much as possible to speed up the application.

3. Analysis Results

As mentioned, we analyze the speech recognition software using three approaches. First, we run the application on different platforms to understand how well the application scale with processor advancement. Speech input is captured in files and ran in batch mode in order to control the quality of the samples. We then use an Intel toolset called Vtune to collect built-in performance counter values [22][23]. Finally we use a full system simulator called SoftSDV together with a cache simulator to study the memory behavior in more depth.

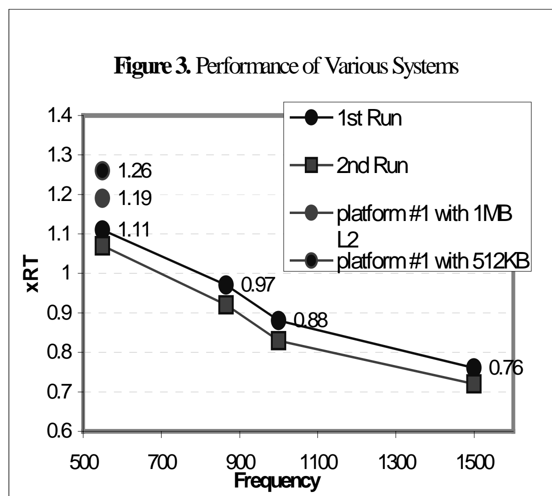


Figure 3. Performance of Various Systems

3.1 Real time performance on different platforms

We use the indicator xRT (real time ratio) to measure the speed of the speech engine. An xRT of 1 means the time to decode is the same as the speech signal time. A lower value of xRT means better performance. We run the LVCSR engine on the following four platforms.

1. Pentium® III 550 MHz with 2MB off-chip L2 cache, 512MB SDRAM. (With 440GX AGPset)
2. Pentium III 866 MHz with 256KB L2 on-chip cache, 512MB RDRAM. (With 840 Chipset)
3. Pentium III1000 MHz with 256KB L2 on-chip cache, 512MB RDRAM. (With 840 Chipset)
4. P41.5GMhz with 256KB L2 on-chip cache, 512MB RDRAM. (With 850 Chipset)

First, each machine is restarted and then the speech engine is run. During this 1st run cache and TLB are all just initialized. After the first run, this LVCSR engine is terminated and is started again. We again capture the running time the 2nd time. Figure 3 illustrates the results of these two runs on the above listed four platforms. The 2nd and the 3rd platforms are identical except the CPU frequency. The 1st platform uses a slower CPU, however the level 2 cache is off-chip and larger in size. The speed of the off-chip cache is half of the core speed, while the on-chip cache has the same speed as the core. Moreover, the 1st platform uses SDRAM instead of RDRAM thus it has a lower memory bandwidth. Two more data points were collected for the 1st platform by switching the CPU with different L2 sizes. It seems a larger L2 even with a slower speed tends to compensate the lack of CPU frequency.

For the 1st platform we further experiment with the memory requirement by varying the amount of the SDRAM in the system and collected the real time rate and CPU utilization rate. Figures 4 and 5 summarize the results. We see both the real time performance and CPU utilization level off after 256MB. Unfortunately we are unable to further delineate between 256MB and 128MB. Having a main memory of 256MB is essential for running this application.

3.2 Computation distribution

Both PIII/P4 processors decode instructions to µ-ops run internally. Figure 6 summarizes the instruction per cycle (IPC) and the micro-ops per cycles (UPC) of different platforms obtained with Vtune™. Even though the 2nd and the 3rd platforms differ only in CPU frequency, their IPC and UPC are different due to memory access.

The rest of this section's analysis is based on the 2nd platform. The computation distribution of running this LVCSR obtained from Vtune™ is shown in

Figure 7 and 8 separated by major tasks in the application. Figure 7 illustrates the execution time distribution among four major tasks while Figure 8 shows the MIPs distribution. Both graphs shows that majority of the time and MIPs is spend on Guassian computation and search.

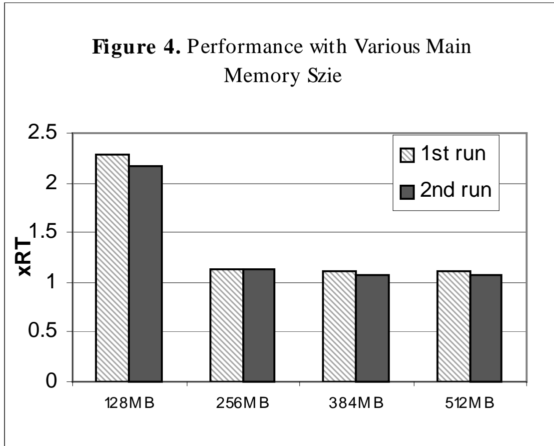


Figure 4. Performance with Various Main Memory Szie

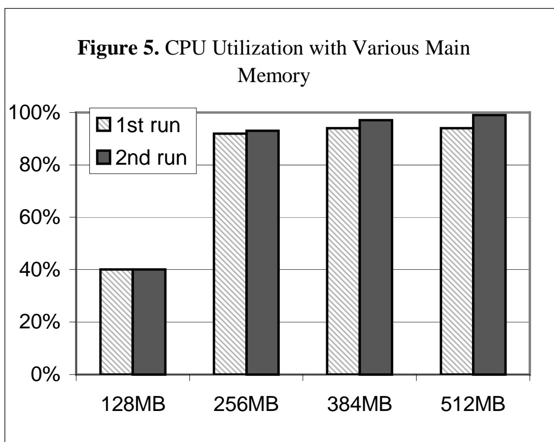


Figure 5. CPU Utilization with Various Main Memory

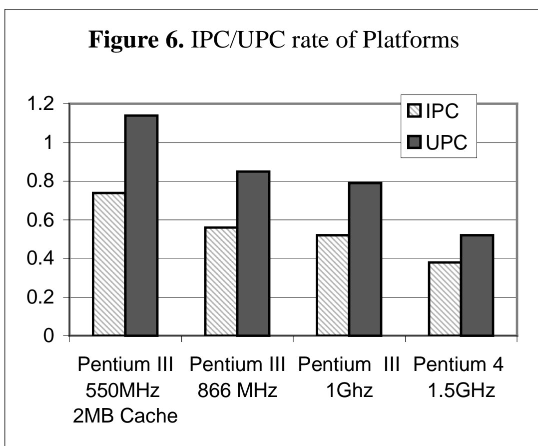


Figure 6. IPC/UPC rate of Platforms

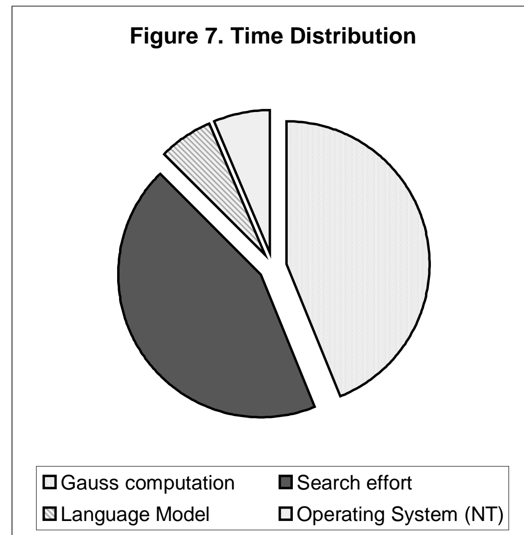


Figure 7. Time Distribution

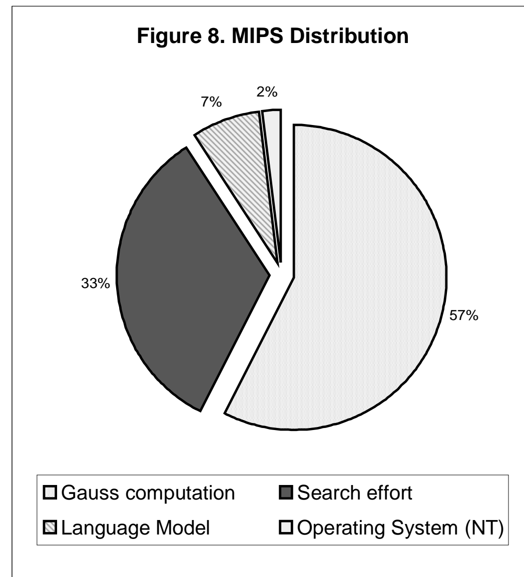


Figure 8. MIPS Distribution

The most frequently seen computation patterns observed are *((a-b)^2)*c* and *log(1+exp(x))*. The later is approximated with a polynomial equation. Both patterns can be expressed with ADD/SUB then Multiply instruction sequences. This is different from the Multiply-ADD/SUB sequences seen in DSP type applications. Usually during the search a great percentage of the actual search time is spend by the Gaussian computation. According to previously reported analysis, Gaussian computation may stall the search process due to dependency. Work done by [17] reports that typically 40%~80% of the total execution time is due to Gaussian Computation. More recently, studies have shown that it is not less than 75% [18]. Our engine, without modification, also reports more than 70% of execution time on Guassian computation. An optimization is used at the algorithm level that speculates the result of computation and reuses results from previously computation. With this optimization

technique we are able to reduce this ratio greatly. It is worthwhile to note, since the cost of the system idle process is counted as OS activity, the actual cost of the language models (include the language look-ahead model) should be higher than the value reported.

When we compare the time distribution with the MIPS distribution we found some small discrepancy. This is because the Gaussian module is composed of more regular and more inherently parallel computations while the search effort is more random. So the time distribution ratio of it is less than the MIPS distribution ratio. The search effort module, also called token propagation module, has many branch and data dependency. Thus, the time distribution ratio of the search module is larger than the MIPS distribution ratio. Unfortunately, these two modules are somewhat interleaved making parallelizing the Guassian module challenging.

There are also some counters for the specific type of instructions. Though they are not detailed enough, some understanding can be obtained. Table 2 lists the break down of instruction types and corresponding miss rates. We will discuss the comparison between LVCSR and other SPEC benchmark programs later.

3.3 Where have all those cycles gone?

We further investigate where have the time been spend in the program. Again all data are collected on the 2nd platform using VTune. Figures 9 and 10 show that about two thirds of the execution cycles are idling and retiring no instructions. This indicates there may be many dependency chains in the application. Note that the maximum number of u-ops can retire each cycle is 3. Since one instruction may be decoded into multiple u-ops, the number of cycles where 3 instructions are retired is less than the number of cycles where 3 u-ops are retired.

Figure 9. Breakdown of IPC

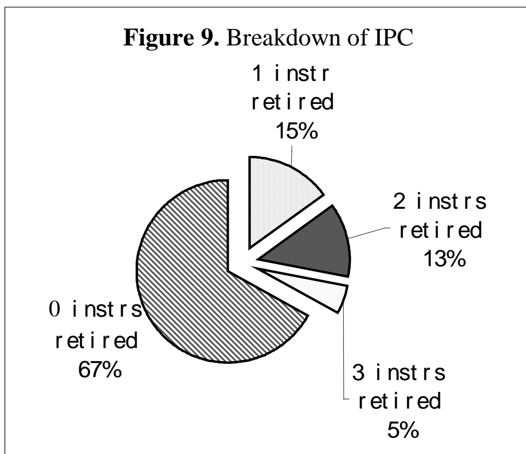
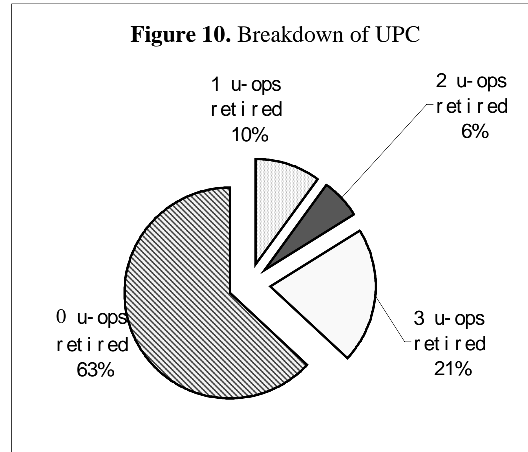


Figure 10. Breakdown of UPC



There are additional counters in Pentium III allowing us to further categorize different types of stalls. Table 1 summarizes the result. The percentage is ranked among all cycles. Accounting for all cycles is difficult in an out-of-order processor such as Pentium III. Execution can proceed during a stall cycle in some other part of the machine. For example, when the pipeline has a resource stall, instruction can continue to be fetched. Similarly, during is a instruction fetch stall, there are other instructions being executed in the other part of the pipeline.

Resource related stalled cycles	52%
Instruction fetch stalled cycles	4.5%
Partial stalled cycles	1.5%

Table 1. Stall cycles

Instruction cache misses and ITLB misses cause instruction fetch stalls mainly. Resource-related stalls are much higher than instruction fetch stalls and partial stalls in LVCSR. This is similar to floating point benchmarks. The resource-related stalls indicate that there are instructions in the application requiring the same hardware resources such as register renaming buffer entries and memory buffer entries [22]. Branch misprediction recovery and delay in retiring mispredicted branches also causes resource-related stalls. Since branch misprediction rate and L2 cache miss rate are not overly high, long dependency chains in the code may have caused these stalls cycles.

At the mean time, there is a large percentage (21%) of the cycles that can retire 3 μ-ops. This indicates that the there are parallelism computations in the program.

3.4 Comparison with SPEC CPU2000 benchmarks

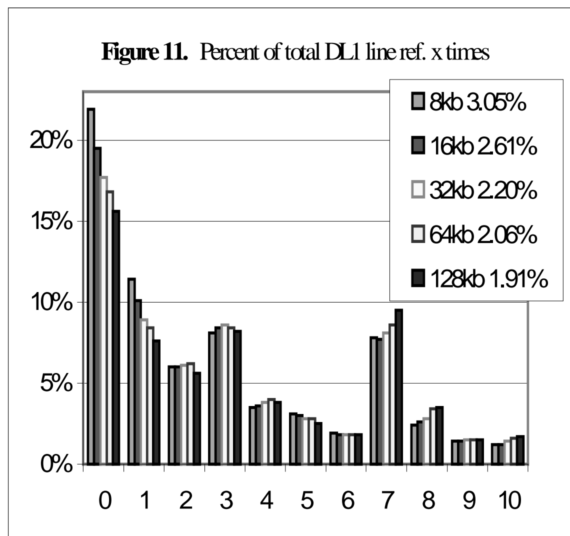
Using the same counters accessible through Vtune™, we collect data for some SPEC CPU2000 benchmarks. Table 2 summarizes the comparison. These SPEC 2000 benchmarks are compiled with Microsoft Visual Studio compiler using default optimization options (-Ox -Zi). Some numbers are not included if they are small and negligible. As mentioned before LVCSR does have large memory requirement and there may have long dependency chains in the program that need data from critical loads. We see the program characteristics of LVCSR speech engine are similar to a typical floating-point application in the SPEC2000[9].

			SPEC 2000 Integer						SPEC 2000 FP		
		lvcsr	gcc	Gzip	eon	parser	perlbmk	vpr	mesa	art	ammp
Execution Aggregate	IPC	0.56	0.41	0.99	0.71	0.57	0.94	0.45	0.64	0.18	0.41
	UPC	0.85	0.75	1.21	1.56	0.86	1.36	0.65	1.12	0.23	0.58
Stall cycles	Resource related	52%	44%	27%	16%	44%	15%	60%	23%	88%	74%
	Partial Stall	1.5%	1.1%		1.9%	11%	3.2%	1.0%	8.4%		
	Instruction Fetch	4.5%	8.5%	1.3%	9.0%		10%	1.6%	1.6%		
Instruction Mixes	Branches	10%	22%	19%	14%	21%	21%	17%	11%	14%	9.6%
	FP	8.5%						5.2%	8.4%	12%	33%
	SSE	16%									
Memory reference/instr.		0.60	0.65	0.40	0.83	0.62	0.51	0.70	0.68	0.58	0.58
Branch mis-prediction		6.4%	7.4%	5.3%	6.1%	5.3%	4.6%	7.9%	2.8%	5.2%	2.3%
L2 instruction fetch/hundred instr.		0.26	0.2	0.03	0.20	0.06	1.6	0.26	0.27	0.12	0.06
Cache miss	DL1	2.8%	4.3%	7.2%	0.14%	2.8%	1.2%	3.8%	0.78%	15%	4.5%
	L2	38%	40%	5.4%	0.27%	35%	6.8%	38%	13%	96%	52%

Table 2. Comparison between LVCSR and SPEC CPU2000 benchmarks

3.5 More detail memory behavior study

As mentioned, besides using Vtune™ we also use software simulation tools to further study the memory behavior. With a built-in first level cache in the SoftSDV's CPU model, we first skip 66.5 billion instructions after starting up SoftSDV. We use 1 billion instructions for warming up cache then collected a 380 million L2 references trace. These traces are used as input to a trace-driven cache simulator. When simulating using this trace we use the first 200 million references to warm the L2 cache.



We first analyze the level 1 cache behavior. The DL1 has relatively good locality. Its miss rate is comparable to "ammp" in the SPEC2000 FP benchmarks. Figure 11 illustrates the cache line reference frequency for various DL1 cache sizes. In this figure the x-axis is the number of times a cache line is referenced before it is replaced. The y-axis is the percentage of lines among all referenced lines. We also show the miss rate for each DL1 sizes in the legend. This behavior is very similar to "ammp". However the L2 cache miss rate of LVCSR is higher than "ammp". There are other SPEC2000 FP benchmarks which act very differently from LVCSR and "ammp".

Figure 12 plots the miss rate verses L2 cache size for various line sizes. Note this is a log/log plot. We see a change of miss rate slope after 8MB. The size requirement is somewhat expected because we know there are three large data storage spaces used for acoustic models, language model and search buffer. Another fact is that our data structure is somewhat large. Each state has 3 to 4 KB of Gaussian parameters. During the search each state is visited one by one. Therefore it is more advantage to have large line sizes. Figure 4 also shows that every time L2 line size is doubled we get similar miss rate as having twice the cache size.

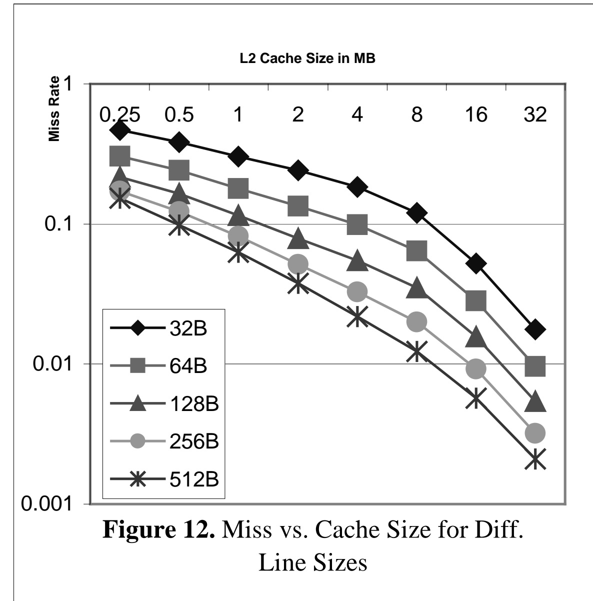
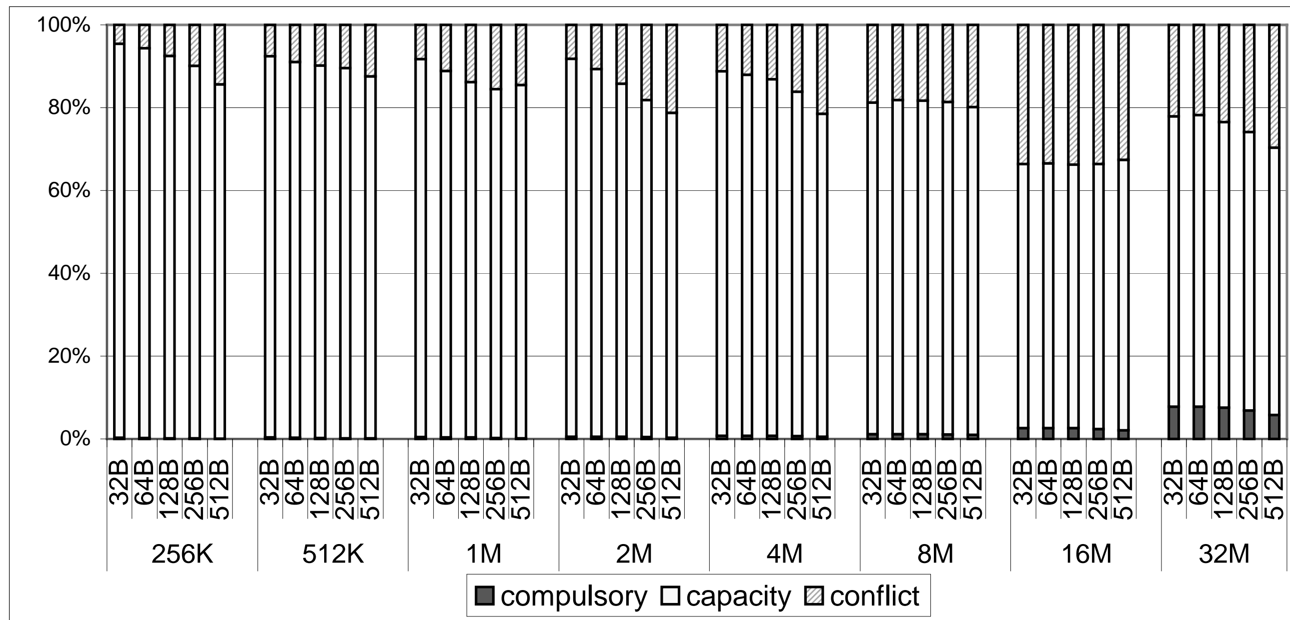


Figure 12. Miss vs. Cache Size for Diff. Line Sizes

Figure 13 shows the capacity ratios of the cache miss with various L2 sizes and L2 line sizes. These data are collected with very long traces. We first skip the first 66.5 billion instructions after starting up SoftSDV. A L1 data cache is built-in with SoftSDV. We use 1 billion instructions to warm up the L1 data cache first. L1 data cache is a typical 32KB 4-way set-associative cache with line size equals to 32B. We then collect around 400 million references to L2. We use about half of these traces to warm up and start collecting statistic after the warm up period. Of traces used to collect statistic 74% are reads. When L2 cache size is small, most of the misses are capacity misses. Very little conflict misses are observed. That may indicate cache associativity need not to be high if we have smaller cache for LVCSR. The L2 capacity miss ratio shows a small drop after the size reaches 16MB.



When we collected traces, we associated with each data reference in the trace the corresponding instruction pointer (IP) address. When we sort these IP addresses we found about 58 IPs that cause 90% of the L2 misses. Moreover, one top IP address causes more than 40% misses (more than 36% of all non-compulsory misses). Figure 14 shows the distribution and accumulated miss rate for these 58 IP addresses. By inserting software prefetch for this one instruction we are able to improve the overall application performance.

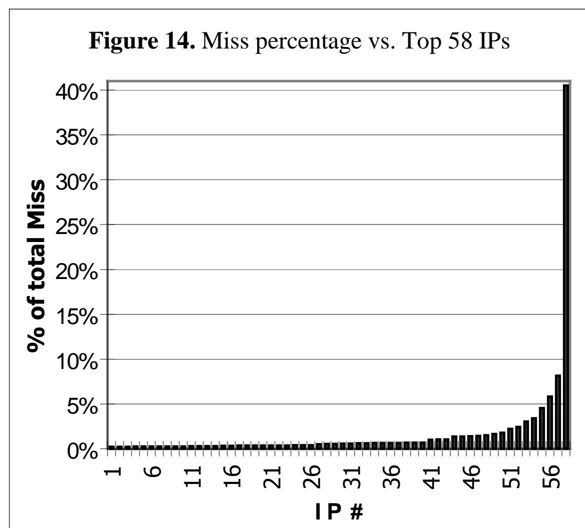


Figure 14. Miss percentage vs. Top 58 IPs

Using traces for cache study is faster but have limitations. In the future we will interface the full system simulator with a cycle by cycle timing accurate memory subsystem model. With that we will be able to collect performance directly instead of estimating the performance impact from miss rates.

4. Conclusion

LVCSR systems based on continuous HMM are gaining usage. We use several tools available to us to study the general characteristic of a LVCSR developed by ICRC. Our study indicates that this LVCSR requires a large number of floating-point computations to evaluate GMM. Since a HMM state uses a large data structure (which typically is 3KB~4KB), there is good data spatial locality. However accesses to language model and acoustic model are random and thus exhibit little temporal locality. This may be one of the main causes of stalled cycles. We also use software simulation to narrow down on instructions that cause most of the level 2 cache misses and use software prefetching to improve performance. Further memory behavior study on the long trace collected will be performed.

Acknowledgement

We thank Konrad Lai, Hubert Hum and John Shen for their discussion and comments. Thanks also to the anonymous reviewers for their many helpful suggestions.

4. References

[1] http://www.dragonsys.com

[2] http://www-4.ibm.com/software/speech

[3] http://www.lhs.com/voicexpress

[4] http://www.speech.be.philips.comhttp://simos.stanford.edu/

[5] K.Agaram, S.W.Keckler, and D.C.Burger. "A Characterization of Speech Recognition on Modern Computer Systems", *4th IEEE Workshop on Workload Characterization*, at MICRO-34, December, 2001 , 2001.

[6] http://fife.speech.cs.cmu.edu/sphinx

[7] Qingwei Zhao, Zhiwei Lin, Baosheng Yuan and Yonghong Yan, "Improvements in search algorithm for large vocabulary continuous speech recognition", ICSLP, Vol.4, October, 2000, Beijing, pp306-309.

[8] http://developer.intel.com/software/products/vtune/index.htm

[9] SPEC CPU2000, http://www.specbench.org/osg/cpu2000

[10] R. Uhlig et. al., "SoftSDV: A Pre-silicon Software Development Environment for the IA-64 Architecture," Intel Technology Journal, 4th quarter, 1999. http://developer.intel.com/technology/itj/q41999/articles/art_2.htm

[11] R. Haeb-Umbach, D. Geller, and H. Ney. Improvements in connected digit recognition using linear discriminant analysis and mixture densities. In ICASSP, pages 239--242.

[12] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," Readings in Speech Recognition, Kaufmann, 1990, pp. 267-296.

[13] S.Young A review of large-vocabulary continuous speech recognition. IEEE Signal Magazine, 1996, Sep, 45-57

[14] X. D. Huang, Y. Ariki, and M. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

[15] E. L. Bocchieri. Vector quantization for the efficient computation of continuous density likelihoods. In ICASSP, pages 692--694.

[16] H. W. Huang, M. Y. Hon, M. Y. Hwang and K. F. Lee, "A Compartive Study of Discrete, Semicontinous, and Continuous Hidden Markov Models," Computer Speech and Language, 1993, No. 7, pp. 359-368.

[17] P Beyerlein, et. al., "Hamming distance approximation for a fast log-likelihood computation for mixture densities". Proc. of the European Conf. on Speech Communication and Technology, Madrid, Spain, vol. II, pp.1083-1086.

[18] S. Ortmanns, et. al., "Language-model look-ahead for large vocabulary speech recognition", ICSLP, 1996, pp2095-2098.

[19] Y.Yan, X. Wu, J. Schalkwyk, R. Cole. "Development of CSLU LVCSR: The 1997 D ARPA HUB4 Evaluation System". In Proceedings *DARPA '98 BNTUW*, 1998

[20] Intel Corporation, Intel Integrated Performance Primitives, http://developer.intel.com/software/products/ipp

[21] Intel Corporation, Intel IA-32 architecture software developer's manual, http://developer.intel.com/design/pentium4/manuals

[22] D. Bhandarkar and J. Ding, "Performance Characterization of Pentium® Pro Processor," Proc. of Symp. On High Performance Computer Architecture, Feb 1-5, 1997, San Antonio pages

[23] Yong Luo, Kirk W. Cameron, Josep Torrellas, and Yan Solihin, "Performance modeling using Hardware Performance Counters", Tutorial, HPCA-6, Jan 2000, Toulouse, France.