Profiling Large-Vocabulary Continuous Speech Recognition on Embedded Devices: A Hardware Resource Sensitivity Analysis

Kai Yu, Rob A. Rutenbar

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, USA
{kaiy, rutenbar}@ece.cmu.edu

Abstract
When deployed in embedded systems, speech recognizers are necessarily reduced from large-vocabulary continuous speech recognizers (LVCSR) found on desktops or servers to fit the limited hardware. However, embedded hardware continues to evolve in capability; today’s smartphones are vastly more powerful than their recent ancestors. This begets a new question: which hardware features not currently found on today’s embedded platforms, but potentially add-ons to tomorrow’s devices, are most likely to improve recognition performance? Said differently – what is the sensitivity of the recognizer to fine-grain details of the embedded hardware resources? To answer this question rigorously and quantitatively, we offer results from a detailed study of LVCSR performance as a function of microarchitecture options on an embedded ARM11 and an enterprise-class Intel Core2Duo. We estimate speed and energy consumption, and show, feature by feature, how hardware resources impact recognizer performance.

Index Terms: speech recognition, software performance, hardware profiling

1. Introduction
Speech is an ideal input modality for resource-constrained environments, so a great deal of effort has been expended to implement speech recognition in mobile devices. However, these devices have limited processing capability due to power, size, and price constraints. Embedded recognizers necessarily pare back features from large-vocabulary, continuous speech recognition (LVCSR) codes found on workstations to achieve usable performance. These changes, like smaller acoustic models, limited language models [1] or semi-continuous HMMs [2], improve recognition speed but also reduce accuracy, sometimes dramatically [2].

Technology trends suggest quality compromises for the sake of speed will become less prevalent in the future. The computational capabilities of embedded devices have greatly improved, to the point where processors found on high-end smartphones now resemble single core workstation processors, adding functionality like the ability to decode multiple instructions per cycle and floating-point operations. Future hardware improvements may allow best-quality speech recognizers to be ported to mobile devices with less attenuation/reduction of critical features. Still, it is unclear exactly how/where recognition performance might be impacted by potential hardware upgrades. Our goal in this paper is to answer these questions in a rigorous way.

When designing an embedded speech recognizer one normally chooses algorithms and models that balance the speed and accuracy constraints against a fixed set of hardware resources. In this paper, we turn this equation around: the recognizer is fixed, but the hardware is variable. We use modern architecture analysis techniques (i.e., expensive cycle simulation) across a wide range of both realistic and idealized hardware mechanisms. We measure the sensitivity of recognizer performance to a large set of possible hardware improvements we may see in the near future.

To extract the sensitivity of speech recognition to hardware, we profiled a typical LVCSR engine on a variety of processor configurations based on an ARM11 core [3-4] and an Intel Core2Duo [5]. We measured both execution time and energy consumption, the two metrics which most affect mobile recognition. From these we determine not only which hardware resources have the largest effect on run time and energy, but also some general guidelines on how to optimize for performance. To our knowledge, this is the first study that (1) measures the sensitivity of recognizer performance across a range of modern architectural features; (2) compares these sensitivities across both embedded and enterprise hardware; and (3) extracts sensitivities for not only recognizer execution time, but also energy consumption.

This paper also helps optimize speech recognition by determining the application's resource utilization and limitations of the current hardware environments. This is especially important in the mobile space due to the large variety of processors, memory storage devices, and batteries. Some decisions are simple, like changing arithmetic to fixed-point when no floating-point is available, but questions like how cache sizes impact recognition speed or what recognizer is the most power hungry, require deeper analysis.

Although there have been previous efforts to characterize speech recognition on processors, the recognizers studied were primitive [6-7], or sacrificed significant accuracy for speed [8]. The previous work also only tested processor architectures found in workstations, so their conclusions are at best a weak fit to embedded devices, whose resource constraints lead to a different set of performance bottlenecks.

2. Profiling Methodology
Processing time and energy consumption are the two essential performance metrics in the mobile domain. Thus we profiled speech recognition using both metrics to determine how the processor architecture affects performance.

To measure processing time and find architectural bottlenecks, we used Simplescalar [9], a cycle-accurate simulator for program performance analysis. We chose to model the ARM11 processor with a floating point coprocessor to simulate the computing resources of a very powerful mobile device and a single core of an Intel Core2Duo to represent a workstation processor. Important SimpleScalar values for both baseline processor configurations are listed in Table 1.

Total power consumption is the sum of the dynamic and static (leakage) power of the CPU. To estimate dynamic
We chose to profile Sphinx 3.0 [13], a large-vocabulary, speaker-independent, continuous speech recognizer from Carnegie Mellon University. Sphinx 3.0 can be divided into three separate stages: feature extraction, feature scoring, and search. Feature extraction takes the input speech and computes 13 MFCCs for every 10 ms of speech. Next, feature scoring uses the MFCCs to generate a 39-dimensional feature vector using the MFCC and its first and second time derivatives and computes probabilities of tied-triphone states represented with Gaussian mixture models (GMMs). Tied-triphone state probabilities are computed for eight frames at a time to increase the amount of computation performed per triphone state. The acoustic model and perform many computations. Although search has very little arithmetic, it is constantly accessing the active HMMs and language model, so it requires considerable memory bandwidth. Any performance optimization must focus on the feature scoring or search, so the rest of this paper will focus on these two stages.

### 3.1. Baseline Configuration

The execution time and energy consumption breakdown for Sphinx 3.0 for the baseline ARM11 and Intel configuration are shown in Table 2. The bulk of the execution time is spent in feature scoring and search, which is consistent with the results from earlier, simpler recognizers. Feature extraction is composed of many digital signal processing algorithms that require very few operations and memory accesses. Feature scoring takes a long time because it needs to read in a large acoustic model and perform many computations. Although search has very little arithmetic, it is constantly accessing the active HMMs and language model, so it requires considerable memory bandwidth. Any performance optimization must focus on the feature scoring or search, so the rest of this paper will focus on these two stages.

### 3.2. Execution Time Bottlenecks

To better understand what limits recognizer decoding speed on the ARM11 processor, we also evaluated Sphinx 3.0 with a combination of “perfect” memory, “perfect” instruction fetch/decode/issue (“perfect” instruction), and “perfect” ALUs. Removing these resource constraints allowed us to see which of them has the largest impact on execution time. To explore a wide range of hardware and architecture options. But total runtime here was a serious concern. The WSJ 5K corpus represents a compromise between LVCSR realism and our ability to complete this study. The experiments shown herein required roughly 3 months of CPU time to collect.
achieve perfect memory, we configured SimpleScalar to have one-cycle latencies for all cache and off-chip memory accesses. For perfect instruction and ALU, we set the number of instructions fetched/decoded-issued and number of ALUs to the largest values SimpleScalar would accept (64 fetches, 16 decodes/issues, 8 integer ALUs, 8 integer multipliers, 8 floating point ALUs, 8 floating point multipliers). Figure 1 shows the cycle count of the two most important stages and the total time normalized against the baseline model results.

Figure 1: Effects of “perfect” memory, instruction, and ALU on ARM11 execution time, y-axis starts at 0.6.

Memory latency and instruction issue have the greatest impact on execution time. It is intuitively obvious why memory is important, because both feature scoring and search require reading data sets which are too large to store on-chip. Instruction issue generally does not limit performance of applications—with large amounts of parallelism, but as discussed earlier the scalar, in-order architecture limits the instruction-level parallelism exploited. Without enough instructions being issued, the functional units remain idle, so “perfect” ALU shows no improvement. When examining individual stages, one finds feature scoring benefits more from instructions issued because of its inherent parallelism, while search favors faster memory because it requires fewer operations and much more memory bandwidth. For the WSJSK corpus, improving memory latency has a greater impact on overall run time, but for other test sets that spend a larger fraction of execution time in feature scoring, run time will benefit more from using a superscalar processor.

Embedded speech recognizers need to run near real-time, so it is unlikely that for this corpus the ARM11 processor will be fast enough without major architectural changes. Even with “perfect” all configuration the best performance is only 21.4% faster than baseline or 7.39 $\times$RT. Better branch prediction alone will not solve the problem. Even with perfect branch prediction, the “perfect all” configuration only improves by 1.82%.

### 3.3. Effects of Cache Size

To see how sensitive performance is to the L1 cache size, we varied the IL1 and DL1 size from 4KB to 64KB. Instead of just using execution time and energy consumed to evaluate the processors, we also used the energy-delay product (EDP) [15], a metric used widely in the low-power digital design community. Any architectural improvement that improves decoding time may also increase energy consumed, so we need a metric that measures the trade-off. By multiplying the energy consumed and decoding time we place equal weight on energy and time, but one can easily add exponents to the variables if one metric is more important.

The effects of L1 cache sizes on execution time, energy consumed, and EDP are shown in Figure 2. The size of IL1 has virtually no effect on execution time because the miss rate for a 4KB IL1 is already only 0.19%. Since larger caches draw more power without improving execution time, it is not surprising to see the EDP is monotonically increasing with IL1 cache size.

Conversely, EDP is parabolic with respect to DL1 size because the large change in execution time from a 4KB DL1 to an 8KB DL1. The improvement lies in the feature scoring stage, and we found it was because the working set does not fit in a 4 KB cache without replacement. Ideally the acoustic model of a single GMM and set of eight feature vectors would fit in DL1 without conflict to prevent time-consuming main memory accesses. However, the malloc function allocated 4KB per GMM so evaluating a single feature vector required off-chip accesses. Since the working set does not fit in a 4KB DL1, there is a large improvement in execution time when increasing the DL1 to 8KB. Further increases in DL1 size lead to larger increases in energy than decreases in execution time so the optimal DL1 size for EDP is 8KB.

### 3.4. Advanced Processor Configurations

Since embedded speed recognition needs a more powerful processor, in the following experiments we measured the sensitivity of performance to reasonable upgrades to the ARM11 architecture. We focused on these four modifications to bring the ARM11 processor closer to the Intel one: (1) 256 KB UL2, (2) double the instructions fetched/decoded/issued, (3) out-of-order (“OOO”) core, and (4) L1 caches with LRU replacement. The resulting recognition speeds, and percent changes are shown in Table 3.

Table 3. Performance of ARM11 processor configurations.

<table>
<thead>
<tr>
<th>Modification</th>
<th>xRT</th>
<th>xRT</th>
<th>ΔEnergy</th>
<th>ΔEDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>9.40</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>w/double instruction</td>
<td>8.58</td>
<td>-8.7%</td>
<td>-4.9%</td>
<td>-13.1%</td>
</tr>
<tr>
<td>w/OOO core</td>
<td>7.86</td>
<td>-16.3%</td>
<td>-7.1%</td>
<td>-22.3%</td>
</tr>
<tr>
<td>w/UL2</td>
<td>9.17</td>
<td>-2.4%</td>
<td>+2.1%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>w/LRU L1 caches</td>
<td>9.39</td>
<td>-0.4%</td>
<td>-0.2%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>w/OOO + 2X inst.</td>
<td>5.11</td>
<td>-45.7%</td>
<td>-25.2%</td>
<td>-59.4%</td>
</tr>
<tr>
<td>w/OOO + 2X inst. + UL2</td>
<td>5.00</td>
<td>-46.8%</td>
<td>-23.9%</td>
<td>-59.5%</td>
</tr>
<tr>
<td>w/OOO + “perfect” all</td>
<td>3.53</td>
<td>-62.5%</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Both doubling the instructions fetched/decoded/issued and switching to an out-of-order (OOO) core greatly improves recognition speed because these changes allowed more of the instruction-level parallelism to be exposed. Doubling the instructions fetched/decoded did not impact performance; all the gains were due to issuing two instructions instead of just one. When the working set is in the DL1 there is enough data and ALUs, all that is missing is the instructions. Using an OOO core leads to a 16% improvement in execution time, more than any other modification or single “perfect” configuration. Both feature scoring and search achieve
individual gains of over 13% because independent instructions are no longer stalled due to memory misses. Although these modifications increase the power the processor draws, the overall energy consumed decreases because the execution time decreased by a larger factor.

Adding the unified L2 cache slightly improved execution time but the gains are offset by the extra energy consumed and increase in chip area. For feature scoring if the working set did not fit in the DL1 cache having a faster cache than main memory would help, but the 16 KB DL1 cache is large enough to prevent thrashing. Other data like the acoustic model and active HMMs are streamed in with little reuse so there is little temporary locality for the UL2 to exploit. Using caches with LRU replacement had a minimal effect.

We also combined the different improvements to see what performance could be expected in the future. Using an out-of-order core capable of issuing two instructions per cycle improved recognition speed by 45.8%, which is greater than the sum of the individual speedups. Its performance is even better than the baseline “perfect” all configuration, even though it has much slower memory, less instructions fetched/issued/decoded, and less ALUs. The two upgrades are complimentary because issuing more instructions helps when the data is in the cache and the out-of-order helps more when the processor needs to wait for main memory. Adding a UL2 cache further improves recognition speed, and its performance is 42% slower than best possible performance an OOO core can achieve mainly due to main memory latencies.

Another way to improve performance is using faster processors, but power is directly proportional to clock frequency so it is unlikely embedded systems will reach the speeds of workstation processors. Also, faster clock frequencies do not completely translate to faster recognition because memory latencies do not scale. Using a 4 times faster clock with our baseline ARM11 configuration improves recognition speed by only 2.51 times to 3.74xRT, and assuming energy scales linearly with frequency the energy used increases by over a factor of 5.

4. Analysis and Conclusions

By profiling Sphinx 3.0 on a resource-limited processor, we understand how hardware resources impact the speed and energy characteristics of a workstation recognizer. Although different recognizers, different acoustic/language models, and different hardware will change the specific breakdowns, the general trends will still hold true.

When designing a recognizer for a specific platform or choosing a platform for a specific recognizer, it is important to find the right balance of memory bandwidth, functional units, and instructions fetched/decoded/issued to optimize the speed and energy. For our baseline system, adding more functional units did not improve execution time and only increased energy consumed, IL1 size matters little, DL1 must be large enough to compute GMM probabilities without capacity or conflict misses, and adding a UL2 cache does not provide enough speedup to offset the power and area costs.

Even with technology scaling, it is doubtful that the ARM11 architecture will achieve workstation-level accuracy or achieve sizable gains in recognition speed unless an out-of-order core is used. The in-order core severely limits the instruction-level parallelism exploited, restricting any further gains from other hardware improvements like issuing more instructions per cycle or adding more on-chip memories. Using an out-of-order core also leads to more energy-efficient speedups than increasing the clock frequency. If further performance improvements are desired, one could remove the limitations inherent to software running on processors by using custom hardware speech recognizers, which can simultaneously improve decoding speed and energy used without impacting accuracy [16].

Of course, in any practical implementation scenario, the recognizer architecture is not fixed, and is adjusted to match the hardware platform. But by reversing these assumptions, we gain some deep insight into how future platforms might evolve – or perhaps be encouraged to evolve – to ease the task of designing the best possible embedded recognizers.

5. Acknowledgements

This research was supported by the National Science Foundation and the FCRP Focus Center for Circuit & System Solutions (C2S2). K. Yu is supported by a National Science Foundation Graduate Research Fellowship.

6. References