Open Source WFST tools for LVCSR cascade development

Josef R. Novak, Nobuaki Minematsu, Keikichi Hirose
Graduate School of Information
Science and Technology
The University of Tokyo
{novakj,mine,hirose}@gavo.t.u-tokyo.ac.jp

Abstract

This paper introduces the Transducersaurus toolkit which provides a set of classes for generating each of the fundamental components of a typical WFST-based ASR cascade, including HMM, Context-dependency, Lexicon, and Grammar transducers, as well as an optional silence class WFST. The toolkit further implements a small scripting language in order to facilitate the construction of cascades via a variety of popular combination and optimization methods and provides integrated support for the T^3 and Juicer WFST decoders, and both Sphinx and HTK format acoustic models. New results for two standard WSJ tasks are also provided, and the toolkit is used to compare a variety of construction and optimization algorithms. These results illustrate the flexibility of the toolkit as well as the tradeoffs of various build algorithms.

1 Introduction

In recent years the Weighted Finite-State Transducer (WFST) paradigm has gained considerable popularity as a platform for Automatic Speech Recognition (ASR). The WFST approach provides an elegant, unified mathematical framework that can be utilized to train, generate, combine and optimize the many heterogenous knowledge sources that typically make up a modern Large Vocabulary Continuous Speech Recognition (LVCSR) system. This has lead to the development of several excellent general purpose software libraries devoted to the construction and manipulation of WFSTs, including the popular open source OpenFst C++ toolkit. Much research has also been conducted on the theoretical construction, integration and optimization of WFST models for ASR (Mohri, 1997; Mohri, 1999; Mohri, 2002; Allauzen, 2004; Mohri, 2008). Nevertheless to our knowledge at present there is no open source toolkit devoted to the construction of ASR-specific WFST models.

This lack of available tools represents an obstacle to the wider dissemination and adoption of WFST-based methods. In response to this, the current work introduces the Transducersaurus WFST toolkit (Novak, 2011), which aims to provide a unified, flexible and transparent approach to the construction of integrated WFST-based ASR cascades, while incorporating recent research results on this important topic. It includes a set of classes for constructing component models as well as a simple Domain Specific Language (DSL) suitable for specifying cascade integration and optimization commands. It provides integrated support for HTK (Young, 2006) and Sphinx (Walker, 2004) acoustic models and cascade construction support for both the T^3 (Dixon, 20007) and Juicer (Moore, 2005) WFST decoders. Where in past complicated development was required, with this toolkit input knowledge sources and a single command are sufficient to build a high-performance system. In addition to introducing the toolkit, this work contributes new experimental results for two LVCSR tasks from the Wall Street Journal (Paul, 1992) (WSJ) corpus, and provides discussion of alternative cascade build chains.

The remainder of the paper is structured as follows. Section 2 describes the main component models of a typical WFST-based ASR cascade. Section 3

Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing, pages 65–73,
Blois (France), July 12-15, 2011.

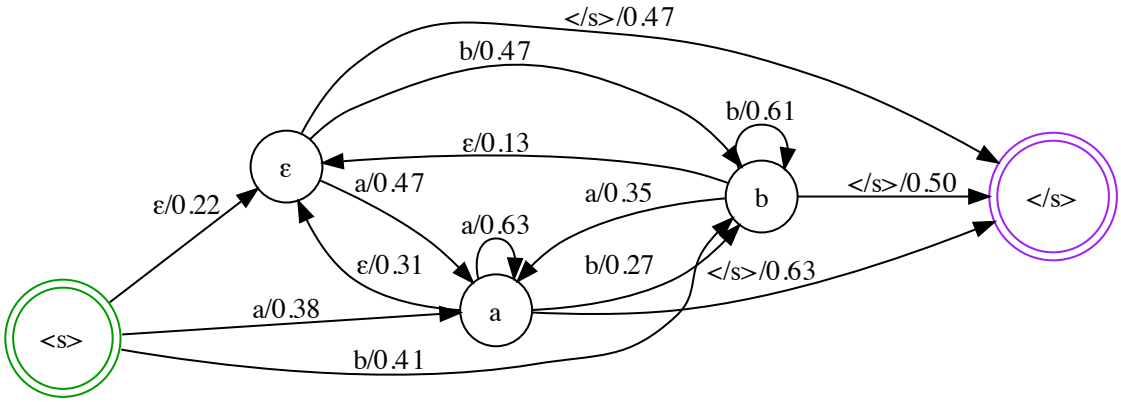

Figure 1: Detail of a bi-gram model for a simple two word LM.

describes the cascade integration tool and its capabilities. Section 4 describes new experimental results that explore the flexibility of the Transducersaurus toolkit. Section 5 provides additional analysis and explores the practical implications of various construction techniques. Finally, Section 6 concludes the paper.

2 Integrated LVCSR Cascades

The construction of WFST-based cascades for LVCSR tasks typically involves two major steps. The first step is to construct WFST-based representations of each of the component knowledge sources, and the second step is to integrate these components into either a single static cascade or, in the case of on-the-fly composition a smaller subset of integrated models. The most common component knowledge sources involved in the first step include a grammar **G**, in the form of a statistical language model, a pronunciation lexicon **L**, that maps monophone sequences to words, and a context-dependency transducer **C**,that maps context-dependent triphone sequences to corresponding monophone sequences. In addition to these three fundamental components, an HMM-level model **H**, that maps HMM state sequences to context-dependent triphone sequences is frequently utilized, and class-based silence models are also popular. The Transducersaurus toolkit provides integrated support for each of the **H**, **C**, **L**, **G**, and **T** component transducers, and these components are described in detail in the following subsections.

2.1 Grammar acceptor

The grammar component **G**, encodes information about word sequences, and typically represents a standard ARPA format statistical N-gram model. Several different approaches to transforming an N-gram model into an equivalent Weighted Finite-State Acceptor (WFSA) have been proposed in the literature (Allauzen, 2003). The simplest approach utilizes a single historyless back-off state, and uses normal ϵ-transitions to encode back-off arcs and associated back-off weights. This is the approach utilized currently in the Transducersaurus toolkit, and a small example of such a model is depicted in Figure 1.

The use of normal ϵ-transitions however, can lead to situations where back-off N-gram sequences may be less costly than the equivalent N-gram sequence. Strictly speaking this is incorrect, and (Allauzen, 2003) discusses two strategies for dealing with this problem. The first involves the use of special "failure" or ϕ-transitions for the back-off arcs. These ϕ-transitions encode the idea that the back-off arc should only be utilized in the event that an equivalent normal N-gram arc does not exist. The second strategy involves mutating the baseline ϵ-back-off configuration, adding additional back-off states and manipulating the back-off arcs so as to eliminate instances of path ambiguity. Transducersaurus utilizes the ϵ-transition approach mainly for the sake of simplicity, but support for the alternative strategies is planned for future work. The toolkit provides a python program, **arpa2fst.py** which may be used

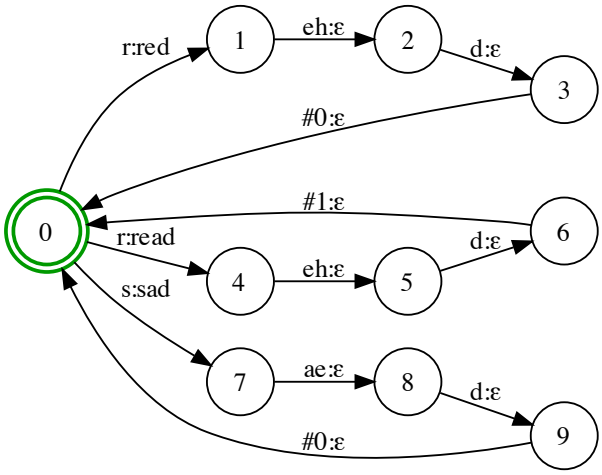

Figure 2: Example of a three-word lexicon transducer, L.

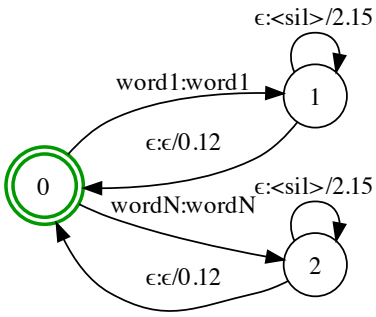

Figure 3: An N-word silence class model, **T**.

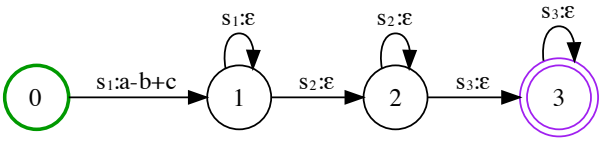

Figure 4: Example of a deterministic three-state HMM model for the triphone *a-b+c*.

to transform a standard ARPA format LM into an equivalent WFSA. The tool also generates symbol tables as needed.

2.2 Lexicon transducer

The lexicon transducer **L**, maps monophone sequences or pronunciations to words. An example of a trivial lexicon transducer is described in Figure 2. In order to ensure that the lexicon can describe not just isolated words, but also word sequences, it is necessary to perform the closure of the resulting WFST prior to downstream composition. Furthermore, in order to handle the occurrence of homophones in the lexicon, it is necessary to augment the construction with auxiliary symbols as described in (Allauzen, 2004). If this step is not taken, the lexicon as well as any downstream cascades may become non-determinizable. The toolkit provides a lexicon generation tool in the form of **lexicon2fst.py**, and this tool supports closure, and auxiliary symbol generation natively. **lexicon2fst.py** provides support for generating HTK as well as Sphinx format lexicons, the latter of which typically utilizes positional triphones. The tool further generates necessary symbol tables, a list of monophones, and a list of any auxiliary symbols that are added during construction.

2.3 Context-dependency transducer

The Context-dependency WFST **C**, maps context-dependent triphone sequences to corresponding context-independent monophone sequences. There are several methods of building this component as well, which are described and illustrated in detail in (Allauzen, 2004). The Transducersaurus toolkit implements a deterministic construction algorithm which results in a C transducer where the output symbols are delayed. There are two separate tools for building the **C** transducer, **cd2wfstHTK.py** and **cd2wfstSphinx.py** and as the names indicate, the first tool provides native support for the HTK format acoustic models, and the second provides native support for Sphinx format models. The **C** tools take as input a list of monophones, an optional list of auxiliary symbols, and an optional acoustic-model specific tied-list. The output consists of the text-format WFST and associated symbol tables. Both tools also provide support for an additional auxiliary WFST which can be used to replace auxiliary symbols or translate logical triphones to physical triphones found in the input acoustic model. This is important in situations where the user wishes to perform further optimizations on a CLG or $HCLG$ cascade.

2.4 Silence class transducer

As with most of the cascade components, there are several viable approaches to handling silence in a WFST-based LVCSR cascade. The Transducer-

saurus toolkit supports a special silence class transducer that can be utilized to transform a grammar by augmenting it with silence or filler arcs. Other alternatives include adding additional silence-trailing entries to the lexicon or utilizing forced-alignment to insert silences into existing speech transcripts. In the latter case the aligned transcripts can then be used directly to build an N-gram model with silence tokens. In the toolkit, the **silclass2fst.py** program can be used to generate a silence class transducer from a list of words. An example of the silence class transducer is depicted in Figure 3. Unlike the lexicon-based approach, the T approach permits long silences, and unlike the N-gram based approach, it encodes the idea that silences may follow any word without a context-sensitive penalty or boost. The trade-off between the silence loop and return ϵ-arc may be specified by the user but the toolkit supplies default values that were estimated from several hundred hours of spontaneous English conversation transcripts.

2.5 HMM level

The HMM-level transducer **H**, maps HMM state sequences from an acoustic model to context-dependent triphones. The toolkit currently focuses on a 3-state HMM configuration, although there are plans to extend this in future to more flexible configurations. An example of a deterministic, three-state **H** WFST for a single triphone, *a-b+c* is depicted in Figure 4. In practice the full **H** transducer describes the closure of the union of all triphones and monophones in the acoustic model. The structure is similar to the lexicon transducer, however the phonemes are replaced with HMM states, the words are represented by monophone and triphone labels, and the length of each entry is fixed to the number of HMM states used to train the models. In most acoustic models such as those produced by HTK and Sphinx, state-tying is used to share HMM states for under-represented models. With the above approach this can lead to non-determinism due to some triphones sharing the same underlying state sequences. This problem is handled by Transducersaurus by adding a second level of auxiliary symbols to the **H** transducer in order to guarantee determinizability. At present the **H** construction tool, **hmm2wfst.py** provides native support for Sphinx format *mdef* files, as well as support for the native AT&T text format. Native support for the HTK *hmmdefs* file format is also underway. Finally, the T^3 decoder provides online simulation of the HMM state self-loops, which eliminates the need to explicitly generate these during construction. Self-loop arc generation is however supported as an option.

3 Cascade integration with Transducersaurus

In most cases it is necessary to first combine the individual models described in the previous sections before they can be utilized for speech recognition. Much work has been done in the past in regards to theoretically optimal cascade optimization and compression methods, for example (Allauzen, 2004) describes several effective composition and optimization schemes and the impact that these have on WACC and decoding speed. Nevertheless the behavior of different construction and optimization schemes can vary considerably based on the size and complexity of the input models. The proposed toolkit provides a cascade integration tool, **transducersaurus.py** the aim of which is to facilitate learning and speed up the potentially tedious and time-consuming process of cascade generation. This tool calls the individual model construction classes described in Section 2 and automatically performs all required generation, compilation, integration and optimization algorithms. The tool further supports a wide selection of common features of WFST cascade generation including semiring selection, auxiliary symbol support, and fundamental WFST operations such as $composition$, $determinization$, and $minimization$ via the OpenFst library. The tool further provides integrated support for both HTK and Sphinx acoustic models. The flagship contribution of this toolkit however, is a simple WFST-oriented DSL which aims to streamline the specification of build algorithms and optimization procedures. This DSL is described in detail in the following section.

3.1 Cascade construction DSL

The DSL supported by the build tool allows the user to specify a build chain using a subset of the standard FST-based combination and optimization algo-

Table 1: A trivial cascade build command demonstrating several of the options available.

```
$ ./transducersaurus.py --tiedlist tiedlist --hmmdefs hmmdefs
--grammar my.lm --lexicon my.lex --amtype htk --convert tj
--command "min(det(H*det((C*det(L)).(G*T))))"
```

rithms, as well as shorthand for the component models described earlier. The user need only specify a simple chain for example,

```
--command "min(det(C*det(L*(G*T))))",
--command "(C*det(L)).(G*T)"
```

and the build tool will automatically tokenize and parse the command into the appropriate series of OpenFst commands, generating intermediate results as necessary along the way. At present the DSL is quite limited, but supports the min, det, $\circ$ (specified "$*$" on the command line) and "." operations as well as the construction of the **H**, **C**, **L**, **G**, and **T** component transducers. Here det refers to determinization, min to minimization, "$*$" to standard composition, and "." to Static Look-Ahead (SLA) composition, which was released in a recent version of OpenFst, and which implements the Look-Ahead composition algorithm proposed in (Allauzen, 2009). Auxiliary symbol replacement is handled automatically in a manner dependent on the set of build commands issued by the user.

The advantage of the DSL approach is that it permits very simple specification of the build chain, which in turn encourages experimentation and learning, and lends itself easily to further extension through the future addition of other standard operations. Thus the user only needs to prepare the component knowledge sources, and specify a build algorithm. For example the command in Table 1 will automatically construct an integrated recognition network utilizing a silence class model, SLA composition and an HTK-format acoustic model, and output an optimized $HCLGT$ cascade suitable for use in both Juicer and T^3.

4 Experiments

The proposed toolkit can be used to generate recognition networks for a variety of different tasks and inputs. In order to showcase this flexibility, several different experiments were carried out making use of different build chains and two test sets from the WSJ corpus. A selection of recent results are reported for HTK and Sphinx acoustic models and bot the Juicer and T^3 decoder. These results illustrate the correctness of the toolkit in reproducing previous baselines, and also confirm separate results encouraging the SLA-based build chains.

4.1 Experimental setup

All experiments for this work were performed on an 8 core Intel Xeon based machine running at 3GHz with a 6MB cache and 64GBs of main system memory running the RHEL OS. As with our previous results from (Novak, 2010), the experiments covered two popular tasks from the WSJ corpus. The first task, *nov92-5k*, focuses on the November 1992 ARPA WSJ test set which comprises 330 sentences, and was evaluated using the WSJ 5k non-verbalized vocabulary and the standard WSJ 5k closed bigram language model. The second task, *si_dt_s2-20k*, focuses on a subset of the WSJ1 Hub2 test set which comprises 207 sentences. The *si_dt_s2-20k* task, which is somewhat more difficult, was evaluated using a 64k vocabulary an a large 3-gram LM trained on 222M words from the CSR LM-1 corpus (Doddington, 1992). In order to help ensure the repeatability of our experiments, open source Sphinx and HTK acoustic models described in (Vertanen, 2006) were used throughout, and auxiliary parameter values for the T^3 and Juicer decoders were specified as in (Novak, 2010). Unless otherwise specified the log semiring was used for all constructions.

4.2 *Nov92-5k* LVCSR Experiments

The first set of experiments focused on the standard WSJ *Nov92-5k* test set, the default closed bigram language model and associated pronunciation lexicon. Open source Sphinx format acoustic models were used. The toolkit was utilized to generate six different cascades, which shared the same fundamental knowledge sources but differed in terms of the optimization procedures applied, and whether an

Table 2: WSJ-based WFST cascade characteristics for Sphinx acoustic models. Here min refers to minimization, det refers to determinization, the "$\circ$" operator refers to standard composition, and the "." operator refers to static look-ahead composition.

Cascade constructions	***Arcs***	***States***	***Size***
$(C \circ det(L)).G$	1,828,710	620,711	36 MB
$det((C \circ det(L)).G)$	3,588,184	726,782	64 MB
$min(det((C \circ det(L)).G))$	3,260,139	654,008	58 MB
$det(H \circ ((C \circ det(L)).G))$	4,226,328	2,729,896	96 MB
$det(H \circ det((C \circ det(L)).G))$	6,981,130	3,528,195	147 MB
$min(det(H \circ det((C \circ det(L)).G)))$	6,318,302	3,107,984	132 MB

H-level transducer was utilized in the cascade. Recent work such as (Allauzen, 2010) as well as our own recent experiments have shown that SLA composition, which omits the $det(LG)$ operation, performs equally well, thus SLA composition was utilized in all six cascade constructions.

The command used to generate these cascades was specified as

```
$ ./transducersaurus.py --tiedlist mdef
--amtype sphinx --grammar bcb05cnp-2g.arpa
--lexicon bcb05cnp.dic --convert t
--base auto --prefix bcb05s
--command "(C*det(L)).G"
```

and the value of the `--command` parameter was simply modified to generate each of the six different variations. The properties of each of the resulting cascades are described in detail in Table 2. The variation in terms of the number of arcs, states and total size clearly indicates the relative effects of applying different optimization operations to the construction process. The simplest construction, $(C \circ det(L)).G$ results in the smallest cascade in this case. Subsequent application of determinization increases the initial size of the cascade, while miniminization again reduces the overall size. This pattern is repeated with the addition of the HMM-level WFST. The set of Sphinx format cascades generated with the **transducersaurus.py** tool were subsequently evaluated inside of the T^3 decoder and the results of these evaluations are described in Figure 5.

Although small in this simple task, the effect of optimization techniques can nonetheless still be clearly seen in the difference between the $(C \circ det(L)).G$ construction and the determinized and minimized variants. In general the impact of these optimizations is increased for larger and more complicated models. Nevertheless the gains are not achieved without a cost. In particular each additional call to the determinization and minimization algorithms consumes significant additional computing resources and time. These requirements grow rapidly as the size and complexity of the input models increases. Thus it is pragmatic to strike a balance between development time, resource requirements and achievable RTF versus WACC. In order to help illustrate this trade-off we also looked at the memory consumption versus time characteristics of the $det(LG)$ determinization operation, the standard $C \circ (LG)$ composition, and the SLA composition, $CL.G$. The results for the bcb05 cascade used to evaluate the *Nov92-5k* test set are depicted in Figure 6, and unequivocally show that the determinization operation is by far the most costly, but also indicate the advantages of SLA composition over the standard variant.

4.3 *si_dt_s2-20k* LVCSR Experiments

The second set of experiments involved the *si_dt_s2* test set, and a much larger 3-gram language model based on the CSR corpus. These experiments were carried out to show that the toolkit is a viable choice not just for small models, but can be used in a straightforward manner to also build very large, efficient cascades. This experiment also illustrates the ability of the toolkit to generate both HTK, and Sphinx based recognition networks and to construct working cascades for both Juicer and the T^3 decoder. In this case experiments focused on a single construction scheme; the simple yet effective $(C \circ det(L)).G$ and the commands utilized to build the cascades are described in Table 4 and informa-

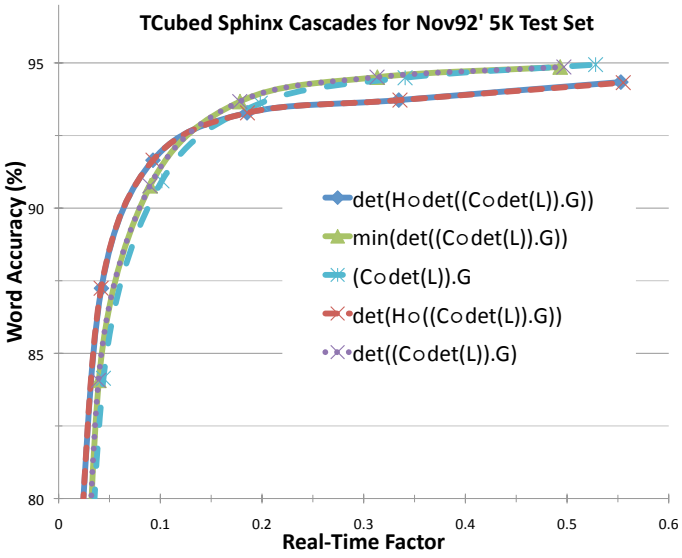

Figure 5: Cascade build comparison for the Nov92-5k task using the T[3] and Sphinx format acoustic models.

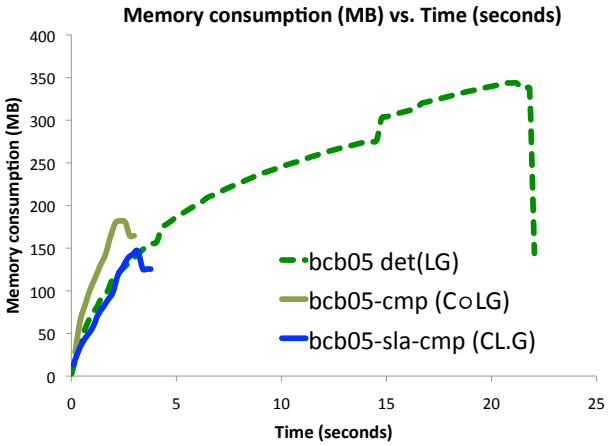

Figure 6: Memory consumption versus time comparison of the $det(LG)$ operation and SLA versus standard composition for the bcb05 CLG cascade.

tion regarding arc and state counts as well as overall size is described in Table 3 while RTF versus WACC results for the three tests are illustrated in Figure 7.

5 Discussion

The results from the two experiments provide new empirical evidence supporting previous research results in this area. Results from Subsection 4.2 show that the toolkit can be utilized to quickly and simply develop a variety of different LVCSR cascades and that build results accurately and reliably reflect previously reported findings. We note that the $HCLG$ builds converge more slowly, but achieve the same best WACC at approximately 2x real-time. The SLA composition algorithm is an improvement over standard composition (Allauzen, 2010), but the most substantial gains from the alternative $(C \circ det(L)).G$ build chain result from the ability to avoid the otherwise costly $det(LG)$ determinization operation in a simple CLG construction. In the experiments described in Subsection 4.2, using SLA composition provided roughly a 50% memory savings, and an average overall time savings of nearly 80%. The cross-comparison results described in Subsection 4.3 replicate previous results from (Novak, 2010), this time utilizing the SLA build. Notably, in this case the SLA build produces significantly smaller cascades and furthermore the relative sizes of the Sphinx versus the HTK format models is reversed. The latter result is likely a consequence of the positional triphones utilized by the Sphinx mod-

Table 3: CSR-based WFST cascade characteristics for HTK and Sphinx models. Both cascades employed a $(C \circ det(L)).(G \circ T)$ construction scheme.

Cascade	*Arcs*	*States*	*Size*
CSR-64k-HTK	146.4M	92.4M	3.3GB
CSR-64k-Sphinx	143.8M	88.8M	3.2GB

Table 4: Cascade build commands for the si_dt_s2-20k LVCSR experiments.

```
$ ./transducersaurus.py --tiedlist mdef --amtype sphinx --grammar
lm_csr_64k_nvp_3gram.arpa --base auto --lexicon lm_csr_64k_nvp.dic
--prefix bcb05s --convert t  --command "(C*det(L)).G"

$ ./transducersaurus.py --tiedlist tiedlist --hmmdefs hmmdefs
--amtype htk --grammar lm_csr_64k_nvp_3gram.arpa  --base auto
--lexicon lm_csr_64k_nvp_3gp-htk.dic --prefix csr64kh
--convert t --command "(C*det(L)).G"
```

els, which permit a smaller degree of sharing, thus resulting in a larger increase in size following determinization in the $C \circ det(L \circ G)$ construction. The small performance variation among the AM types and T^3 versus Juicer again suggest that there is not much technical motivation to overtly favor any particular combination. Rather the availability of resources and existing expertise should guide development choices.

Finally, the T^3 decoder also supports GPU-based computation of acoustic likelihood scores, and these results have been reported in several previous works. We note however, that application of GPU-based acoustic scoring, when available tends to provide the strongest single speedup, and that use of the more computationally intensive $logsum$ operation versus the standard $logmax$ also tends to boost maximum accuracy. This implies that SLA composition, combined with GPU-based acoustic scoring and a comparatively simple $(C \circ det(L)).G$ build chain provides highly competitive results. This strikes a strong balance between RTF, WACC, memory and storage requirements and overall build time. Further savings in terms of memory requirements, storage and build time can be gained from performing the lookahead composition on-the-fly at decoding time.

6 Conclusion and Future Work

In this work we have introduced Transducersaurus, a new open source software toolkit for building and manipulating WFST-based ASR cascades. The toolkit provides integrated support for the T^3 and Juicer WFST decoders and both HTK and Sphinx acoustic models, and supports construction of the **H**, **C**, **L**, **G**, and **T** component WFSTs. We showed the effectiveness of the toolkit on a variety of different tasks, looking at both construction variants on a simple set of inputs, and performing a decoder and acoustic model cross comparison on a much larger task. Furthermore we have provided a detailed explanation of the SLA build process as it is supported by the toolkit along with its merits. The ASR application development process is often iterative, and these results reinforce the idea that by utilizing a simplified build chain and the SLA composition approach, overall efficiency can be greatly improved at little or no cost to either the RTF or WACC of a particular recognition network.

In future we plan to further expand the range of available operations, and expand the current limited DSL build syntax, provide integrated support for out-of-vocabulary words, and introduce parallel support for the AT&T fsmtools. Experiments looking at a much wider variety of languages and model inputs currently in the planning phase. Although the toolkit is still in the early stage of development we hope that it will facilitate learning as well as more efficient work in this area, and promote further discussion.

At present the Transducersaurus toolkit can be downloaded freely from the location listed in (Novak, 2011), and is available under the terms of the liberal BSD license.

References

Mehryrar Mohri 1997. *Finite-State Transducers in Language and Speech Processing*, in Computational Linguistics, Vol. 23, Issue 2.

Mehryar Mohri and Michael Riley. 1999. *Network optimizations for large-vocabulary speech recognition*, Speech Communication, Vol. 28, Issue 1.

Mehryar Mohri, Fernando Pereira and Michael Riley. 2002. *Weighted finite-state transducers in speech recognition*, Computer Speech and Language, Vol. 16, Issue 1.

Cyril Allauzen, Mehryar Mohri, Michael Riley and Brian

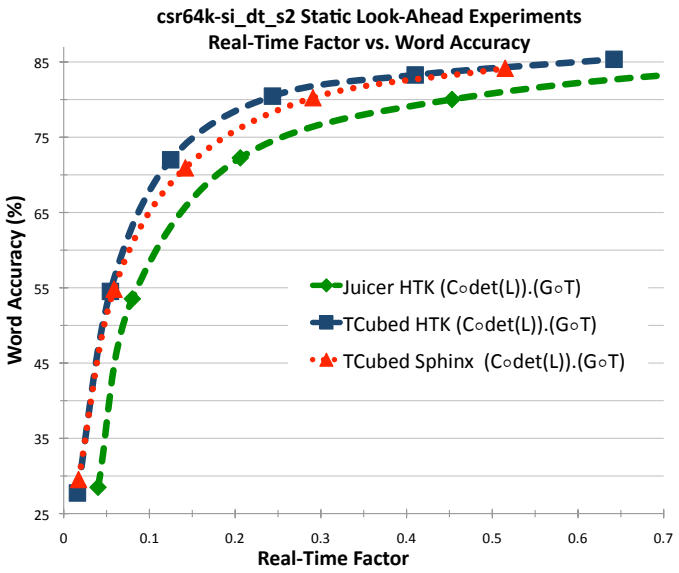

Figure 7: Cross comparison for the si_dt_s2-20k task using the T^3 and Juicer decoders, Sphinx and HTK format acoustic models on the 3-gram CSR LM-1 language model.

Roark. 2004. *A Generalized Construction of Integrated Speech Recognition Transducers*, in Proc. ICASSP, pp. 761-764.

Mehryar Mohri, Fernando Pereira and Michael Riley. 2008. *Speech recognition with weighted finite-state transducers*, Springer Handbook of Speech Processing, pp. 1-31.

Josef Novak. 2011. `code.google.com/p/transducersaurus/`

Steve Young, Gunnar Evermann, Dan Kershaw, Gareth Moore, Julian Odell, Dave Ollason, Valtcho Valtchev and Phil Woodland. 2006. *The HTK Book (for HTK Version 3.2)," Cambridge University Engineering Department*.

Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf and Joe Woelfel. 2004. *Sphinx-4: A flexible open source framework for speech recognition*, Sun Microsystems Technical Report, TR-2004-139.

Paul Dixon, Diamantino Caseiro, Tasuka Oonishi and Sadaoki Furui. 2007. *The Titech Large Vocabulary WFST Speech Recognition System*, in Proc. ASRU, pp. 1301-1304.

Darren Moore, John Dines, Mathew Magimai Doss, Jithendra Vepa, Octavian Cheng and Thomas Hain. 2005. *Juicer: A Weighted Finite State Transducer Speech Decoder*, in Proc. Interspeech, pp. 241-244.

Douglas Paul and James Baker. 1992. *The Design for the Wall Street Journal-based CSR Corpus*, in Proc. ICSLP 92, pp. 357-362.

Cyril Allauzen, Mehryar Mohri and Brian Roark. 2003. *Generalized Algorithms for Constructing Language Models*, in Proc. ACL, pp.40-47.

Josef Novak, Paul Dixon and Sadaoki Furui. 2010. *An Empirical Comparison of the T^3, Juicer, HDecode and Sphinx3 Decoders*, in Proc. InterSpeech 2010, pp. 1890-1893.

Cyril Allauzen, Michael Riley and Johan Schalkwyk. 2009. *A Generalized Composition Algorithm for Weighted Finite-State Transducers*, InterSpeech 2009, pp. 1203-1206.

George Doddington. 1992. *CSR Corpus Development*, DARPA SLS Workshop," pp. 363-366.

Keith Vertanen. 2006. *Baseline WSJ Acoustic Models for HTK and Sphinx: Training Recipes and Recognition Experiments*, Cavendish Laboratory, University of Cambridge.

Cyril Allauzen and Michael Riley. 2010. *OpenFst: A General and Efficient Weighted Finite-State Transducer Library*, tutorial, SLT.