Accelerating Cogent Confabulation: an Exploration in the Architecture Design Space

Qinru Qiu, Daniel Burns, Michael Moore, Richard Linderman, Thomas Renz, and Qing Wu

Abstract— **Cogent confabulation is a computation model that mimics the Hebbian learning, information storage, inter-relation of symbolic concepts, and the recall operations of the brain. The model has been applied to cognitive processing of language, audio and visual signals. In this project, we focus on how to accelerate the computation which underlie confabulation based sentence completion through software and hardware optimization. On the software implementation side, appropriate data structures can improve the performance of the software by more than 5,000X. On the hardware implementation side, the cogent confabulation algorithm is an ideal candidate for parallel processing and its performance can be significantly improved with the help of application specific, massively parallel computing platforms. However, as the complexity and parallelism of the hardware increases, cost also increases. Architectures with different performance-cost tradeoffs are analyzed and compared. Our analysis shows that although increasing the number of processors or the size of memories per processor can increase performance, the hardware cost and performance improvements do not always exhibit a linear relation. Hardware configuration options must be carefully evaluated in order to achieve good cost performance tradeoffs.**

I. Introduction

To build a machine with human intelligence has always been a dream of computer scientists. During the last couple of decades, many cognitive architectures have been proposed. Some are based on explicit logic and expert rules, such as ACT-R [2] and SOAR. Others are connectionist approaches, among which the most dominant models use neural networks. Other recent work integrates advances in neural science with artificial intelligence. In their book [3] Hawkins and Blakesleee advocate that the construction of cognitive architectures should be based on our present understanding of brain function. The Ersatz Brain project [4] is an example of neural science inspired cognitive architecture, which models the brain as a network of cortical columns. Another neural science inspired cognitive architecture is the cogent confabulation model proposed by Hecht-Nielsen [1]. It seeks to mimic the Hebbian learning, information storage, inter-relation of symbolic concepts, and recall operations of the brain.

Most of the previous work on cognitive architectures focuses on cognitive science. They assume that the training speed or the response time of recall is not a concern. However, for any of the above mentioned systems to perform a cognitive application that has meaningful significance, a huge amount of data must be processed quickly. It is estimated that a computer with 100 million megabytes memory and 100 million MIPS may be able to match the human brain [5]. Currently, such a requirement can not be achieved in a uniprocessor system. Parallel processors with multi-core architecture and distributed processing capability are the only feasible solution [6].

This paper presents our research which explores the potential for accelerating the computations that underlie cogent confabulation. Our approach includes both software and hardware optimization. First, we demonstrate the impact of different data structures on the performance of the algorithm. Our experiments show that speedups of more than 5,000X can be achieved by proper selection and tuning of data structures. Then we investigate the potential of accelerating the computations by parallel processing. Analysis shows that cogent confabulation is an ideal candidate for parallel processing. As a model that mimics the behavior of human brain, its computations can be partitioned into small tasks and distributed to different processing elements (PEs). During the analysis, we identified the smallest unit tasks, and carried out extensive software profiling to to model and estimate computational complexity.

Although increasing the number of PEs or the size of their memories can increase computational performance, it also increases system cost. Cost and the performance improvements do not always exhibit a linear relation, i.e. investing 1% more in hardware does not lead to a 1% increase in computational speed. The best implementation scheme should achieve a balance between cost and performance. Given the same cognitive computing model, many different hardware and software implementations are possible. Each has an associated performance (i.e processing delay) and cost (i.e. required resources, # gates, etc.). These implementations are considered design points in the cost-performance space. A design point is Pareto [7] if it has either less cost or lower delay than any other points in the design space. Obviously, only Pareto design points are relevant, and the main idea of architecture exploration is to generate design points and identify those that are Pareto.

In this work, we evaluated various hardware configuration options for implementing the training and recall systems of a single sentence completion confabulation module, and we generated a set of design points with different performance-cost tradeoffs. The efficiency of different systems was measured and compared by computing cost-delay-products.

Most of the previous work on performance optimization of cognitive architectures focuses on hardware or software implementation. For example, a hardware implementation of spiking neural networks is proposed in reference [8]. Reference [9] investigates the architecture design of a Brain-state-in-a-box model. The authors of [10] propose software based optimization of cognitive applications by exploiting their fault-tolerance and noise-tolerance properties. To the best of our knowledge, the present work is the first that studies both software and hardware optimization, and considers performance optimization and hardware cost at the same time.

Manuscript received December 1, 2007.

Qinru Qiu and Qing Wu are with the Department of Electrical and Computer Engineering, Binghamton University, Binghamton, NY 13902 USA (e-mail: {qqiu, qwu}@binghamton.edu).

Daniel Burns, Richard Linderman, and Thomas Renz are with Air Force Research Laboratory, Rome Site, 525 Brooks Road, Rome, NY 13441, USA (e-mail: {Daniel.Burns, Richard.Linderman, Thomas.Renz}@rl.af.mil).

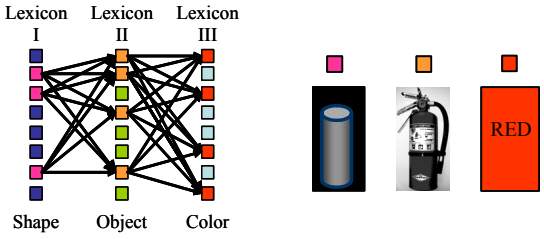
Michael Moore is with ITT Advanced Engineering & Sciences, 775 Daedalian Drive, Rome, NY 13441, USA (e-mail: Michael.Moore@rl.af.mil).

The remainder of this paper is organized as follows: Section II provides necessary background on cogent confabulation theory and the sentence completion example. Section III presents different data structures for software optimization and analyzes the computation complexity and the potential of parallel processing. The cost-performance analysis of different architectures is presented in Section IV. Conclusions are given in Section V.

II. Background

Cogent confabulation is an emerging theory proposed by Hecht-Nielsen. Based on the theory, the information processing of human cognition is carried out by thousands of separate thalamocortical modules. Each of these thalamocortical modules is a patch of cerebral cortex plus a uniquely paired zone of thalamus and is referred to as a *lexicon* or a *feature attractor module*. Different collections of neurons in the thalamocortical module represent different symbols. Knowledge is stored as the links between neurons and their strength. The cognitive information process consists of two steps: learning and recall. During the learning, the knowledge links are established and strengthened as symbols are co-activated. During recall, a neuron receives excitations from other activated neurons. A "winner-take-all" strategy takes place within each lexicon. Only the neurons (in a lexicon) that represent the winning symbol will be activated, and the winning neurons activate other neurons through knowledge links.

Figure 1 shows an example of lexicons, symbols and knowledge links. The three columns in Figure 1 (a) represent three lexicons for the concept of shape, object and color with each box representing a neuron. Different combinations of neurons represent different symbols. For example, as Figure 1 (b) shows, the pink neurons in lexicon I represent a cylinder shape, the orange neurons in lexicon II represent a fire extinguisher, while the red neurons in lexicon III represent red color. During learning, if the information of a red cylinder shaped fire extinguisher is repeatedly provided then the links between these neurons will be strengthened. During recall, if the neurons representing the fire extinguisher are activated, then they will further excite the neurons that represent red color and cylinder shape. If the excitation levels of these neurons are higher than others, then the corresponding symbol will be activated.



(a) A simple example with 3 lexicons and many knowledge links

(b) The pink, orange and red neurons represent 3 symbols

Figure 1 Example of lexicons, symbols and knowledge links

A computation model for cogent confabulation is proposed in [1]. Based on this model, a lexicon is a collection of symbols. A knowledge base (KB) from lexicon A to B is a matrix with the row representing a source symbol in A and a column representing a target symbol in B. The ijth entry of the matrix represents the strength of the link between the source symbol s_i and the target symbol t_j. It is quantified as the conditional probability $P(s_i \mid t_j)$. The knowledge bases are constructed during the learning procedure. During recall, the excitation level of all symbols in each lexicon is evaluated. Let l denote a lexicon, T_l denote the set of lexicons that have knowledge bases going into lexicon l, and S_k denote the set of symbols that belong to lexicon k. The excitation level of a symbol t in lexicon l can be calculated as:

$$I(t) = \sum_{k \in F_l} \sum_{s \in S_k} I(s)[\ln(P(s \mid t)/p_0) + B], \quad \forall t \in S_l . \qquad (1)$$

The function $I(s)$ is the excitation level of the source symbol s. Due to the "winner-takes-all" policy, the value of $I(s)$ is either "1" or "0". The parameter p_0 is the smallest meaningful value of $P(s_i \mid t_j)$. The parameter B is a positive global constant called the *bandgap*. It is introduced to ensure that a symbol receiving inputs from M active knowledge links will always have a higher excitation level than a symbol receiving (M-1) active knowledge links, regardless of the strength of the links.

Based on the example provided in [1], prototype confabulation based single sentence completion software has recently been developed internally at AFRL/RITC [11]. A sentence is represented using 40 lexicons that are arranged in 2 levels. The ith lexicon on level 1 represents the word (or punctuation) in the ith location of a sentence. There are 20 lexicons in level 1, and the words or punctuations beyond the first 20 are discarded. The ith lexicon on level 2 represents the phrase (consisting of one or more words) that begins in the ith location of a sentence. The connections of knowledge bases are "causal". Within each level, each lexicon only establishes knowledge bases with all lexicons to the right. The lexicons in different levels are also connected with knowledge bases. The ith lexicon in level 1 is connected to the jth lexicon in level 2 where $j \le i$ while the jth lexicon in level 2 is connected to the ith lexicon in level 1 where $i \ge j$. Figure 2 shows the lexicons and the knowledge bases for the sentence completion problem.

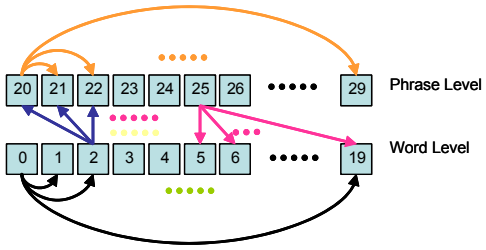


Figure 2 Confabulation based sentence completion

Overall, there are 800 knowledge bases in the system. The contents of the knowledge bases are learned by reading novels and scientific papers that are stored on hard disk. The ultimate size of the knowledge bases depends on the extent of training. Let N denote the total number of unique words, phrases and punctuations in the training corpus. Each lexicon is a collection of N symbols and each knowledge base is potentially an $N \times N$ matrix. A medium sized training file with 173,000 words generates about 60,000 symbols. Extensive software profiling has been performed on this example training file, which will be referred as the *benchmark*.

During learning, each sentence in the training corpus is read in turn. As each sentence is read, each word/phrase is converted into a unique ID based on its location in a master lexicon or dictionary. Each entry in the knowledge base matrix is associated with two symbols. Its row index corresponds to the ID of the source symbol while its column index corresponds to the ID of the target symbol. The knowledge base entry is incremented if the source and target symbols appear in the sentence. At the end of the training, each entry in a knowledge base gives the number of co-occurrences of the corresponding source and target symbols.

After training, the conditional probability of the source and target symbols $P(s \mid t)$ is calculated as: $P(s \mid t) = \frac{kb[s][t]}{\Sigma_i kb[i][t]}$, where $kb[s][t]$ represents the content of knowledge base matrix at row s and column t.

During the recall, given one or more initial words the software is able to generate meaningful sentences that may or may not be in the training text. The starter words are first assigned to the corresponding lexicons, and the excitation level of each symbol in each lexicon is evaluated using equation (1). The symbols with the highest excitation are stored in the activation list. The procedure repeats if the new activation list is different than the previous activation list.

III. Software Optimization

A. *Improving the data structure for the training function*

A knowledge base is essentially an $N \times N$ matrix, where N is the total number of possible words/phrases in the training file and it is about 60,000 for a medium sized training file. Storing all knowledge bases in memory without compression will be impossible. Our experiments show that a knowledge base is a sparse matrix, with about 500 ~ 9,000 nonempty rows and less than 5,000 non-zero entries in those non-empty rows. Therefore, only 1.25% of a knowledge base is non-zero. To compress the knowledge base, we store only the non-zero entries. We optimized the data structure of knowledge bases in a progressive way. Figure 3 shows the four data structures that have been evaluated.

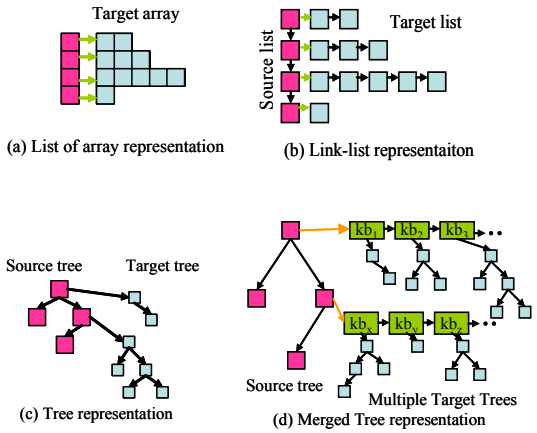


Figure 3 Data structures for KB training

(a) List of arrays. List of arrays data structure is first used to represent each knowledge base matrix. It consists of a source array and a set of target arrays. Each entry in the source array corresponds to a non-empty row in the matrix. It points to a target array which stores the non-zero entries in that row. Figure 3 (a) shows the list of array representation of a knowledge base (KB) matrix with 4 non-empty rows.

(b) Link-list. The list of arrays is not efficient for data insertion and deletion. Each data insertion or deletion requires a memory movement. We replace an array with a link-list. The resulting data structure has one source link-list and a set of target link-lists. Figure 3 (b) shows the link-list representation of the previous example. Compared with the list of arrays based implementation, the link-list based data structure improves training speed by 200X times.

(c) Tree. To locate a knowledge base entry during the training, we need to first find the corresponding row based on the source symbol ID, and then find the corresponding column based on the target symbol ID. The link-list based data structure is not efficient for data search. We further improve the data structure by replacing each link-list with a binary tree. The resulting data structure has one source tree and a set of target trees for each knowledge base matrix. An example of tree based representation is given in Figure 3 (c). Compared with the link-list implementation, the tree based data structure improves the training speed by 4X times.

(d) Merged trees. Close observation of Figure 2 shows that several knowledge bases share the same source lexicon. Therefore, their source tree can be merged to reduce storage and search time. The resulting data structure has one source tree for each lexicon and one target tree for each non-empty row in each knowledge base matrix. Each source node points to an array of knowledge bases that originate from the corresponding lexicon. Each entry of that array points to a target tree. Figure 3 (d) shows the merged tree data structure. Compared with the tree based data structure, the merged tree based data structure provides a 7X speed up for training.

All of the above mentioned performance improvements were obtained using our benchmark training file. Our experiments show that the merged tree is the best data structure for training.

B. *Software profiling*

Software profiling was performed to determine the number and size of the source and target trees (which set memory requirements), and to measure the training speed of the sentence completion system using the benchmark training file with 173,000 words. Table 1 summarizes the software profiling results. Figure 4 (a) and (b) shows the probability distribution of the size of the source trees and target trees, respectively.

Table 1 Summary of tree size

	#of trees	Average size	Max. size	Min. size
Source	40	4026 nodes	8943 nodes	510 nodes
Target	2.78×10^6	3 nodes	4216 nodes	1 nodes

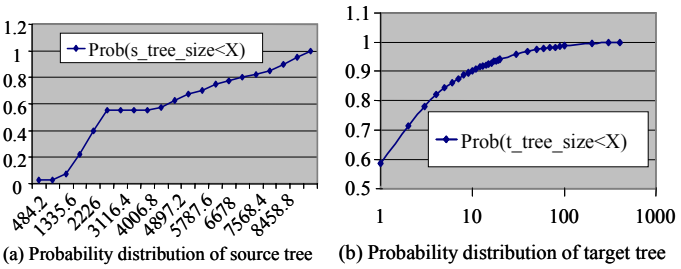


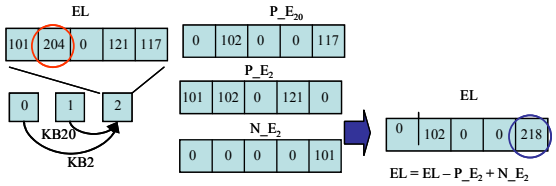
Figure 4 Probability distribution of source and target trees

C. *Improving data structure for the recall function*

During the recall, a partial sentence is input to the system, the excitation levels of all symbols in all lexicons are calculated, and the symbols with the maximum excitations are activated.

Figure 5 shows a simplified example of the recall. Three lexicons (0, 1, and 2) and two knowledge bases (KB2 and KB20) are considered. Each lexicon is a collection of 5 symbols and the knowledge bases are 5×5 matrices. The knowledge bases store the values $\ln(P(s \mid t) / p_0) + B$ with B equal to 100. The contents of KB2 and KB20 are given in Figure 5. Initially, the first symbol in lexicon 0 and the second symbol in lexicon 1 are activated. Therefore, the excitation levels of the 5 symbols in lexicon 2 are calculated as the summation of the first row in KB2 and the second row in KB20. The result is a vector called *Excitation Level* (*EL*) and in this example,

EL is [101, 204, 0, 121, 117]. Assume that later the first symbol of lexicon 0 is deactivated, and the third symbol is activated. The two vectors in the first and second rows of KB2 and KB20 are referred as *Previous Excitation vectors* (P_E_i) while the vector in the third row of KB2 is referred as the *New Excitation vector* (N_E_i). The subscript indicates the index of the knowledge base. The *EL* vector should be updated as $EL - P\_E_2 + N\_E_2$.



(a) The calculation of the excitation level

$$KB\ 2 = \begin{bmatrix} 101 & 102 & 0 & 121 & 0 \\ 111 & 0 & 0 & 105 & 0 \\ 0 & 0 & 0 & 0 & 101 \\ 0 & 0 & 0 & 104 & 101 \\ 104 & 0 & 0 & 0 & 0 \end{bmatrix} \quad KB\ 20 = \begin{bmatrix} 0 & 107 & 0 & 0 & 0 \\ 0 & 102 & 0 & 0 & 117 \\ 0 & 0 & 0 & 0 & 101 \\ 121 & 0 & 0 & 104 & 0 \\ 102 & 101 & 0 & 0 & 0 \end{bmatrix}$$

(b) Knowledge bases KB2 and KB20

Figure 5 A simplified example of recall

The example shows that, for a given activated source symbol, all entries in the corresponding row of a knowledge base must be visited. However, a tree is not efficient for data traversal. The knowledge base matrices are constant during recall, i.e. no data insertion or deletion is performed. Therefore, it is more efficient to replace each target tree in the merged tree data structure with a target array. We refer this data structure as merged tree with target array. An example of a knowledge base that is represented in this data structure is given in Figure 6.

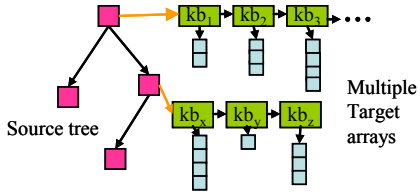


Figure 6 Data structures for Recall

The previous analysis shows that it will be more efficient if training and recall are implemented using different data structures. This would require a special process to convert the knowledge base representation after training. The drawback of doing this is that two representations of the knowledge bases must be maintained at the same time, and that new knowledge cannot be used as soon as it is learned, but only after a process updates the recall knowledge bases.

D. software analysis

1) Training complexity

Figure 7 shows a flow diagram of the training function using the merged tree data structure. It is a repeated procedure consisting of the following steps. At the beginning, a sentence is read in. All the words and phrases in the sentence are converted into their unique ID and assigned to the corresponding lexicons. For each lexicon, the associated source tree is searched to locate the source symbol. Overall, there are 40 lexicons and hence up to 40 source trees will be searched, depending on the length of the sentence. After that, based on the target symbol ID, all target trees that were pointed to by the source nodes are searched and the value of the corresponding knowledge base entry will be incremented. Because there are 800 knowledge bases, up to 800 target trees will be searched. If any of the tree searches returns an empty pointer (i.e. the symbol is not found), a node is created and appended to the end of the tree. This procedure repeats until the entire training file has been processed. (This description is simplified, as the process actually involves counts bases on various source and destination word positions and phrase lengths). The computation complexity is determined by the size of source and target trees. Assume that all trees are balanced. Let avg_n_s and avg_n_t denote the average size of source tree and target tree respectively. During a tree search, 3 operations are required to access a tree node: (1) read the value of the node, (2) compare the value, and (3) read the address of child node. Therefore, each training iteration has an average of $40 \cdot \log(avg\_n_s) \cdot 3 + 800 \cdot \log(avg\_n_t) \cdot 3$ operations. Given a conventional sequential processor, one operation takes at least one clock cycle. For our benchmark training file, where $avg\_n_s = 4026$ and $avg\_n_t = 3$, the sequential processing of one iteration in the training function takes 5240 clock cycles.

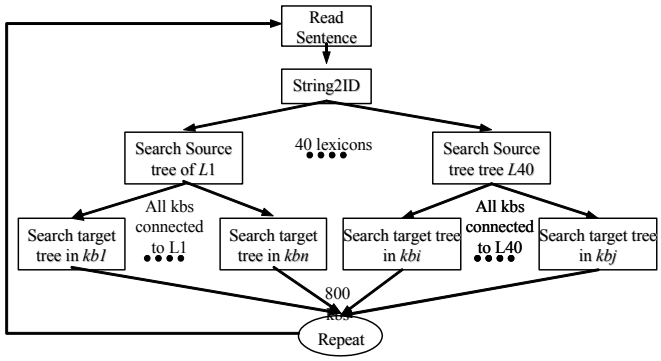


Figure 7 Flow diagram of the training function

2) Recall complexity

Figure 8 shows the flow diagram of the recall. In the first step, if a new symbol is activated in lexicon *l*, then its location *i* is found by searching the source array associated with lexicon *l*. Let $KB_{ll'}$ denote a knowledge base that connects the lexicon *l* to lexicon *l'*. In the second step, the *i*th row of each $KB_{ll'}$ is copied to the new excitation vector ($N\_E_{ll'}$). In the last step, all the new excitation vectors that have been updated during this iteration are added and all the corresponding previous excitation vectors are subtracted to calculate the new *EL*. Then the symbol with the maximum *EL* is found.

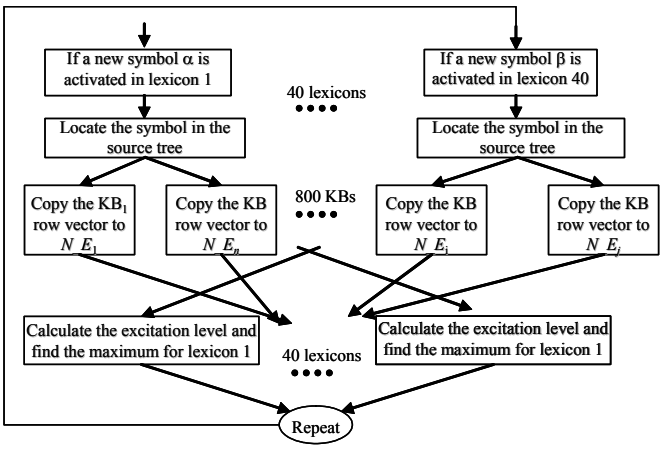


Figure 8 Flow diagram of the recall function

Assume that half of the knowledge bases have a new source symbol being activated. The first step consists of 20 source tree searches which require $20 \cdot \log(avg\_n_s) \cdot 3$ operations. In average, these 20 lexicons connect to 400 knowledge bases. Hence, 400 new excitation vectors will be copied in the second step. This corresponds to 400 * avg_n_t operations. Based on our previous assumption, about half of the knowledge bases that go into a lexicon update their excitation. Therefore, in step 3, about 400 *N_E* vectors must be added to and the same number of *P_E* vectors must be subtracted from the *EL*. This corresponds to $400 \cdot avg\_n_t \cdot 2$ operations. Further more, for each of the 40 *EL* vectors, the maximum value must be found, which takes $40 \cdot avg\_n_s / 2$ operations. Overall, one recall iteration consists of $60 \cdot \log(avg\_n_s) + 1200 \cdot avg\_n_t + 20 \cdot avg\_n_s$ operations. For the benchmark training file where avg_n_t=3 and avg_n_s=4026, it takes 84,838 cycles to process one recall iteration sequentially.

E. Parallel processing in cogent confabulation

As a computation model that is inspired by the neurobiological architectures of the human neocortex, cogent confabulation is an ideal candidate for parallel processing.

Before presenting the parallel model of cogent confabulation, we introduce the concept of an *atomic task*. An atomic task is a sequence of operations which must be applied to the same storage elements. Therefore, these operations cannot be parallelized. Two atomic tasks communicate with each other only at the beginning or end of a process. An atomic task is triggered by another atomic task and it may further trigger other atomic tasks.

Based on the definition, the source tree search (S-search) processes and the target tree search (T-search) processes are 2 types of atomic tasks in the training function. The T-search processes are triggered by the completion of S-search process. The recall function consists of 3 types of atomic tasks. They are: source tree search, *N_E* vector generation, and *EL* vectors calculation.

Atomic tasks of the same type are independent of each other and can be made parallel. For example, during training, up to 40 S-search tasks or 800 T-search tasks can run, simultaneously. The parallel nature of the model can be explored for performance enhancement if provided with appropriate computing hardware.

IV. ARCHITECTURE EXPLORATION

Computation hardware that is able to provide the maximum achievable performance for cogent confabulation should have large number of processing elements (PEs) so that all atomic tasks that belong to the same type can run in parallel. Special hardware such as the *content addressable memory* (CAM) can also be used to improve the performance. However, multiple PEs and special hardware increase the cost of the system. Cost and performance often exhibit non-linear relationships. Careful decisions must be made to in order to find the best tradeoff between cost and performance. In this section, we provide a cost-performance analysis of various hardware architectures designed for confabulation based single sentence completion.

A. Hardware acceleration of the training function

1) Hardware requirements for best achievable performance

The best performance of the training function can be achieved if all atomic tasks that belong to the same type can be made parallel and if each atomic task can be finished in minimum time. The minimum time of S-search and T-search is 2 clock cycles, given the condition that the PEs have large enough CAMs so that the tree search can be finished in 1 clock cycle. For an S-search task, the second cycle is used to trigger the T-search task, while for a T-search task the second cycle is used to update the knowledge base entries. Since the S-search task and its succeeding T-search tasks may very well be mapped onto different PEs, communication latency to pass the triggering event also needs to be considered. However, this can be compensated a little bit by latency hiding techniques such as double buffering. For the best possible performance, we assume that inter-PE communication takes 1 clock cycle. Overall, the best achievable performance for the training function is 2+1+2 = 5 clock cycles.

Architecture (1) Different architectures are able to deliver the best achievable performance. Among them, the most cost efficient system has 840 PEs so that the 40 S-search tasks and 800 T-search tasks can be made parallel. Target trees that belong to the same KB will not be accessed in the same iteration and will be stored on the same PE.

Each PE has a CAM and a RAM. An *n*×*m* CAM is able to store *n* words whose width is *m* bits. When presented with an *m* bit input, the CAM gives an *n* bit output. A "1" at the *i*th location of the output indicates that the *i*th data in the CAM matches the input data. Figure 9 (a) gives an example of a 5×9 bit CAM. When the input is "111111110", which matches the contents in location 1 and 4, the output of the CAM is "10010".

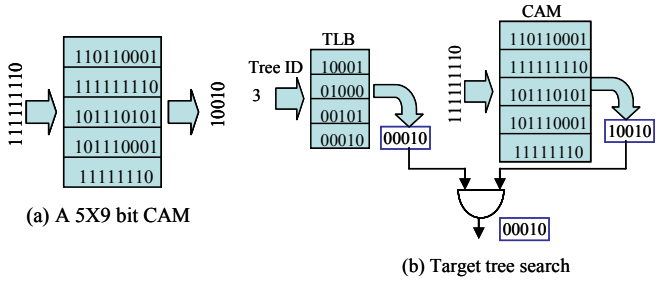


Figure 9 CAM and TLB

When all target trees that belong to the same KB matrix are mixed together and stored in the same CAM, if a matching location is found, a TLB (Table Lookup Buffer) is used to determine whether this location belongs to the target tree of interest. Each entry of the TLB corresponds to a target tree. Each bit of a TLB entry corresponds to a location in the CAM. If the *j*th bit of the *i*th TLB entry is "1", then the *j*th CAM location stores a symbol that belongs to the *i*th target tree. For example, Figure 9 (b) shows the CAM and the TLB of a knowledge base that has 4 target trees (i.e. the knowledge base matrix has 4 non-empty rows.) To search whether symbol "111111110" is in target tree 3, the symbol ID goes to the CAM while the target tree ID goes to the TLB. Since location 1 and 4 in the CAM match the input, the CAM output is "10010". The address 3 of the TLB stores the value "00010", which indicates that target tree 3 has only one symbol and it is stored in CAM location 3. If a bit wise AND operation on the CAM output and TLB output returns a non-zero result, then the symbol is in the target tree. The location of the "1" in the result gives the address of the target symbol. The length and the width of the TLB are determined by the maximum number of target trees and the length of the CAM respectively.

Besides CAM and TLB, a RAM is used to store the pointers or the values of KB entries. Each data entry in the CAM has a corresponding data entry in the RAM.

Table 2 Hardware requirement of Architecture (1)

#PEs	CAM per PE		RAM per PE		TLB per PE	
	Length	Width	Length	Width	Length	Width
840	38K	18bit	38K	32 bit	9K	38 Kb

Using the source and target tree statistics from the benchmark training file, the hardware cost of architecture (1) is estimated and

provided in Table 2. Compared to the sequential process, this architecture provides a 1048X speedup.

2) Performance cost tradeoffs

Architecture (1) in the previous section is very expensive because it requires large number of PEs and each PE must have large amount of memory. In this section we consider lowering the hardware cost by removing certain components, and evaluate the effect on performance. The resulting hardware configurations are not able to provide the best achievable performance; however, they are less expensive.

Architecture (2) For the same storage capacity, a CAM is about 4~5 times larger than a RAM. Therefore, our first option is to reduce the size of the CAM. The system has 840 PEs and a smaller CAM for each PE. The CAM is only large enough to store one source tree. The target trees are stored in the RAM.

Given such system, an S-search task still takes 2 cycles. The system cannot proceed to the next iteration until all of the 800 T-search tasks finish. Therefore, the performance is bounded by the slowest one among the 800 tasks. This is determined by the size of the largest tree among 800 randomly selected target trees. Assuming that special hardware is available to pre-fetch the address of the next node during tree search, it takes 2 cycles to visit each node. The average time to finish all of the 800 T-search tasks is $2 \cdot \log_2 avg\_max800$ cycles, where *avg_max800* is the expected size of the largest target tree among the 800 random samples.

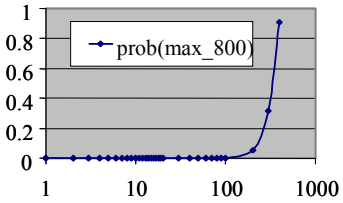


Figure 10 Probability distribution of *avg_max800*

As an example, Figure 10 shows the probability distribution of the size of the largest tree among 800 random samples generated from the benchmark training file. Although the majority of target trees are very small, among 800 random samples, the probability that the largest tree has less than 200 nodes is less than 0.1. The expected size of the largest tree among 800 random samples is 563. The average time to finish all of the 800 T-search tasks is $2 \cdot \log_2 563 = 18$ cycles. Overall, one training iteration takes 21 cycles. Compared to the sequential processing which takes 5240 cycles per iteration, the speed up is about 250X.

Table 3 Hardware cost and performance of architecture (2)

	Performance (cycles)	CAM size	RAM size	# PEs
General	$3+2\log_2 avg\_max800$	max_n_s	$Max(max\_n_s, 4 \cdot max\_n_t)$	840
Bench	*21*	*9K*	*152K*	*840*

The length of the CAM is determined by the largest source tree. Each entry in the CAM must have a corresponding word in the RAM. Besides this, the RAM is also used to store all the target trees. Each tree node is associated with 4 data. They are: {symbol ID, address of the left child, address of the right child, entry in knowledge base matrix}. Therefore, the length of the RAM must be greater than max(max_n_s, $4 \cdot max\_n_t$), where max_n_s is the size of the largest source tree and max_n_t is the maximum number of target nodes in one KB. Finally, because the CAM is not shared among different source trees or target trees, the TLB is not needed. The second row in Table 3 summarizes the hardware requirements and performance for architecture (2). In the third row of Table 3, we replace the variables such as *avg_max800*, max_n_s, and max_n_t with the data that are collected from the benchmark training file and give the estimation of the performance and hardware cost.

Architecture (3) We can further reduce the size of the CAM. When the length of the CAM is less than max_n_s, some source trees cannot be fit into the CAM entirely. The source tree symbols that enter the system earlier will be stored in the CAM. Software profiling shows that the symbols that enter the system earlier have higher probabilities of being visited in the future. Figure 11 shows the relation between the entering order of a symbol and the number of times it has been accessed. Each blue dot in the plot represents a symbol in the source trees. A symbol that enters the system earlier has a lower entering order. The plot shows that symbols with lower entering order will be accessed more frequently. The symbol access frequency is almost inversely proportional to its entering order. The relation can be approximated by the function Access_Times = 1600/Enter_Order, which is the magenta curve in the plot. The observation indicates that the CAM hit ratio will be greater than the CAM storage ratio.

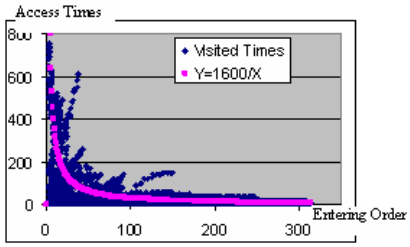
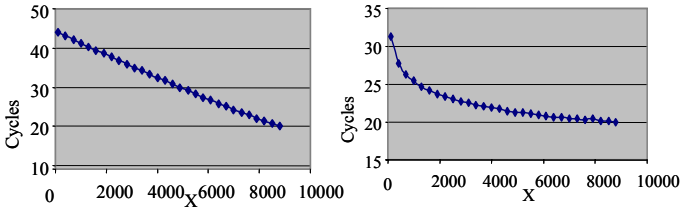


Figure 11 The nodes that enter the system earlier have higher possibility to be visited

If there is a CAM hit (i.e. the target symbol is located in the CAM), then the processing time of the S-search task is 2 cycles. Otherwise, the rest of the source trees must be searched and the system performance is bounded by the search time of the largest source tree. If all symbols have an equal probability of being visited, the expected processing time of the S-search task can be calculated as $T_1 = 2X / \max\_n_s + 2(1 - X / \max\_n_s) \log_2 (\max\_n_s - X)$, where X is the size of the CAM. Similar to previous discussions, we assume that special hardware is available to pre-fetch the address of the next node, so that each node access takes 2 cycles. The expected time for T-search tasks is $2 \cdot \log_2 avg\_max800$ cycles. Figure 12 (a) gives the relation between processing time and the CAM capacity of a system that is capable of processing the benchmark training file, for which max_n_s = 8943 and *avg_max800* = 563.



(a) Process time v.s. CAM size when all symbols have equal access probability

(b) Process time v.s. CAM size when early symbols have higher probabilities of being accessed

Figure 12 Performance estimation

The above analysis is a pessimistic estimation of the system performance. Since the CAM hit ratio is higher than the CAM storage ratio, the actual probability that a symbol is located in the CAM is greater than $X / \max\_n_s$. Assuming that the relation between the number of times a symbol is accessed and its entering order follows the function y=1600/x, and using the profiled statistics,

the relation between the process time and the size of CAM is derived and presented in Figure 12 (b). As we can see, increasing the CAM from 1 to 1500 entries can reduce the processing time from 31 to 24. After that the latency reduction slows down. Since the rest of the source tree must the stored in the RAM, the length of the RAM must be greater than $\max(max\_n_s + 3\times(max\_n_s - X), 4\times max\_n_t)$.

Architecture (4) Reducing the number of PEs can also lower the system cost. Assume that the system has P processing elements. When P < 840, there will be PEs shared by several atomic tasks. Depending on the value of P, the resource sharing can be categorized into 3 types.

(6a) $800 \le P < 840$: Some S-search tasks will be mapped to the same PE as the T-search tasks. Such resource sharing will not impact the performance; however, this requires that each PE be equipped with more memory. Let X denote the size of CAM for each PE, the size of the local RAM of a PE is calculated as: $max\_n_s + 3\times(max\_n_s - X)+4\times max\_n_t$.

(6b) $40 \le P < 800$: T-search tasks that are not mutually exclusive are mapped to the same PE. Therefore, the latency of step 2 of the training function increases.

The average target tree sizes are different for different KBs. Figure 13 shows the profiled information obtained from the benchmark training file. It shows that the KBs that go from level 1 lexicons to level 2 lexicons usually have larger target trees, while the KBs going from level 2 lexicons to level 1 lexicons usually have smaller target trees. In order to maintain a balanced workload among all PEs, the summation of the average tree size of the KBs assigned to the same PE must be approximately equal.

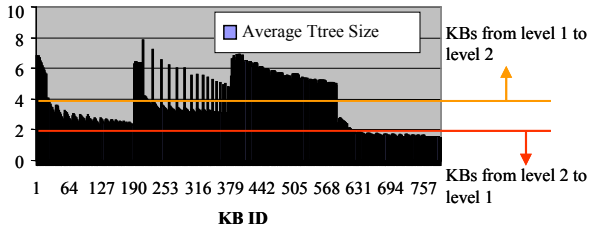


Figure 13 Average size of the target trees for different KBs

(6c) $P < 40$: Different Type I tasks must share the same PE. Figure 14 gives the profiled information for the size of 40 source trees obtained from the benchmark training file. It shows that the source trees associated with the level 1 lexicons are relatively smaller than the source trees associated with the level 2 lexicons. Again the mapping between source trees and PEs must be designed carefully so that the workloads for different PEs are balanced.

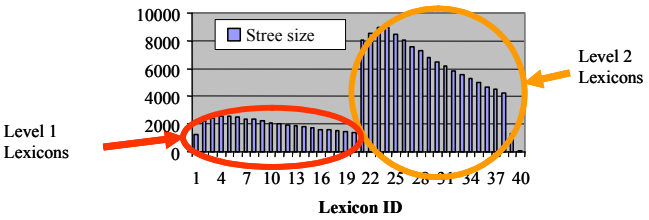
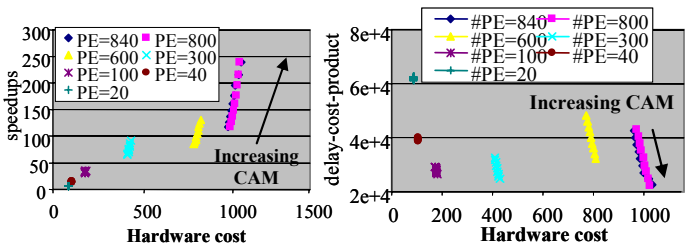


Figure 14 The size of 40 source trees.

Using the benchmark training file as an example, we vary the number of PEs as well as the size of CAM per PE and obtained a set of 73 design points in the cost and performance space. During the analysis, we assumed that the each processor has 1 unit area (i.e 1 unit cost), which is also the area of a 1MB RAM. The CAM is 4 times larger than a RAM with the same capacity. Note that changing the relative size of PE, RAM and CAM will change the results of the following analysis.



(a) Cost vs. speedups (b) Hardware efficiency

Figure 15 Cost-performance analysis for the training system.

Figure 15 (a) shows design points for 73 different hardware configurations in the performance cost space. The X axis represents the hardware cost and the Y axis represents the speedup compared with the sequential process. As we can see, varying the number of PEs gives a larger impact on the cost performance trade off than merely varying the CAM size. A hardware system with more PEs but a smaller CAM for each PE has lower performance and higher cost than a hardware system with less PEs but a larger CAM for each PE. The cost-performance point for the two extreme hardware configurations, i.e. single PE without CAM and 840 PEs with unlimited CAM (i.e. Architecture (1)) are not shown in the plot. The former has 35 units of hardware cost and 1X speedup while the latter has 37,098 units of hardware cost and 1048X speedup.

Figure 15 (b) shows the delay-cost-product (DCP) for different configurations. A lower DCP indicates more efficient hardware. As we can see, increasing the CAM size improves the hardware efficiency. Again, the DCP of the two extreme cases are not shown in the figure. The DCP of a single PE sequential processor is 185,493 while the DCP of the optimal performance hardware is 184,679. It is interesting to note that the DCP of these two extreme cases are very close and they are much higher than the DCPs of the other configurations.

B. Hardware acceleration of the recall function

1) Hardware requirements for best achievable performance

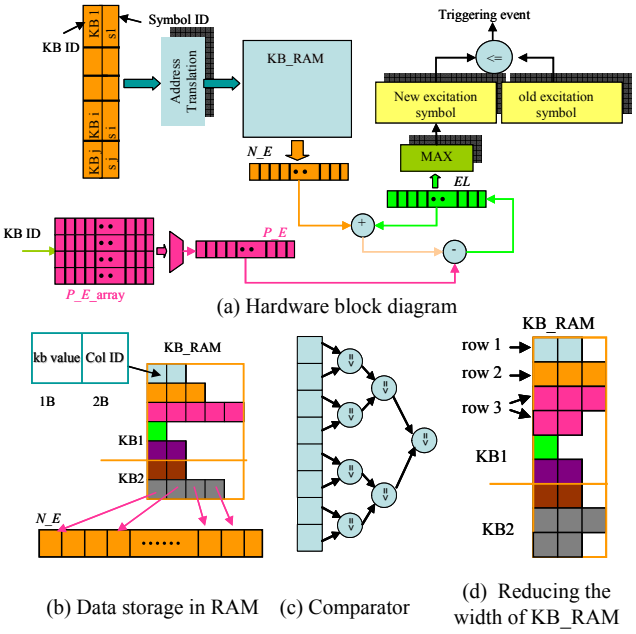
For a lexicon i, let $F(i)$ denote the set of lexicons that connect to knowledge bases coming out from i and $T(i)$ denote the set of lexicons that connect to knowledge bases going into i. The recall system has 40 PEs. Each PE is associated with a lexicon i. It performs three major tasks: (1) receives newly excited source symbols from PEs associated with lexicons j, $j \in T(i)$, (2) calculates the EL vector and finds the symbol with highest excitation, and (3) sends the symbol with highest excitation to other PEs associated with lexicon k, where $k \in F(i)$, if the symbol is different from the previous one. To calculate the EL vector, the PE performs the following 3 steps: (1) reads the N_E/P_E vectors, (2) calculates the EL vectors, and (3) searches EL vector to find the highest excited symbol.

To achieve the highest performance, special hardware is used to pipeline the operations. Figure 16 (a) shows the hardware block diagram of a PE. The hardware has 2 memory blocks: KB_RAM and P_E_array.

The KB_RAM stores all the knowledge bases that go into this lexicon. Each storage element in the RAM consists of two fields, the value of the knowledge base element and the corresponding column index. To achieve the highest performance, each row of the KB_RAM stores one non-empty row of the KB. When a row is read out, based on the column ID, the KB elements are copied into the corresponding locations in the N_E vector. Figure 16 (b) shows how data is stored in the KB_RAM and how it is copied into the N_E vector. A KB_RAM has $\sum_{j \in T(i)} num\_row_{KB_{ji}}$ rows, where

$num\_row_{KB_{ji}}$ represents the number of non-empty rows in the knowledge base that goes from j to i. Each row of the KB_RAM has $\max_{j \in T(i)}(max\_row\_len_{KB_{ji}})$ elements, where $max\_row\_len_{KB_{ji}}$ is the length of the longest non-empty row in the knowledge base that goes from j to i, and each element has 3 bytes. Each row in the P_E_array is associated with a knowledge base KB_{ji}, where $j \in T(i)$, and it stores the previous excitation vector (P_E) of the corresponding knowledge base.

Each PE has an input array that stores the newly excited source symbol from other lexicons. Each entry in the input array has two fields: the knowledge ID (KB ID) and the symbol ID. Based on this information the corresponding row in the KB_RAM is read out and copied into the N_E. Address translation is necessary. However, since the size of the knowledge bases are fixed, the address translation can be implemented using combinational circuits. Based on the KB ID, the P_E vector is read out from the P_E_array. Vector addition and subtraction is performed on EL, N_E and P_E and the result is written back to EL. The addition and subtraction is chained and can be completed in one clock cycle. The two operations, i.e P_E/N_E read and EL calculation are pipelined. The throughput of the pipeline is 1 vector per clock cycle.



(a) Hardware block diagram

(b) Data storage in RAM (c) Comparator (d) Reducing the width of KB_RAM

Figure 16 Hardware block diagram of PE for recall function

To find the maximum value of the EL in short time, a comparator tree is used. Figure 16 (c) shows the comparator tree. The tree has $\log_2 max\_row$ levels, where max_row is the maximum number of non-empty rows in all knowledge bases. There are max_row-1 comparison operations in the tree. It takes $\lceil \log_2 \max\_row \times t_{comp} / T_{clk} \rceil$ cycles to complete the search, where T_{clk} is the clock period and t_{comp} is the delay of a comparator. Because the comparison operations in different levels are mutually exclusive, they can share the same hardware comparator. Therefore, each PE has $max\_row/2$ comparators.

Assume that the inter-PE communication takes only 1 clock cycle. Because in average there are 20 KBs that will have an updated source symbol, to read out N_E/P_E and calculate EL takes 20+1 cycles. The last cycle is to flush out the pipeline. The time to find the maximum value in EL is $\lceil \log_2 \max\_row \times t_{comp} / T_{clk} \rceil$ cycles. Overall, one recall iteration takes $22 + \lceil \log_2 \max\_row \times t_{comp} / T_{clk} \rceil$ cycles.

When the PEs are homogeneous, the size of the RAM and P_E_array should be set to consider the worst case scenario. For a recall system that is based on the knowledge learned from the benchmark training file, the minimum size of the KB_RAM is 217,000×27,000=5.8G byte and the minimum size of the P_E_array is 40×9K=360K byte. It requires 4,500 8-bit comparators and 18,000 8-bit adders. Assume that the T_{comp} is 1ns while the T_{clk} is 2ns, with the above mentioned PE configuration, each iteration takes 29 cycles. Compared to the sequential implementation, the speedup is 2924X.

2) Cost performance tradeoff

The above configuration is very expensive because the width of the KB_RAM is bounded by the longest row in KB and each PE must be equipped with 5.8GB memory. We reduce the width of KB_RAM to W bytes, where $W < 3 \times \max_{j \in T(i)}(max\_row\_len_{KB_{ji}})$. For those KB rows with more than W/3 non-zero values, we wrap them around and store them in multiple RAM entries. For example, in Figure 16 (d), row 3 is wrapped around and occupies 2 RAM entries. This reduces the wasted storage space, however at a cost of increased latency. Multiple reads are required to fetch some of the long rows.

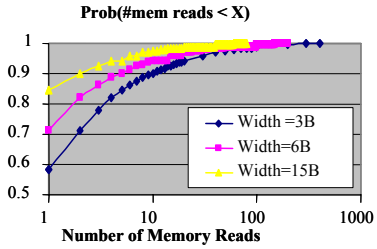


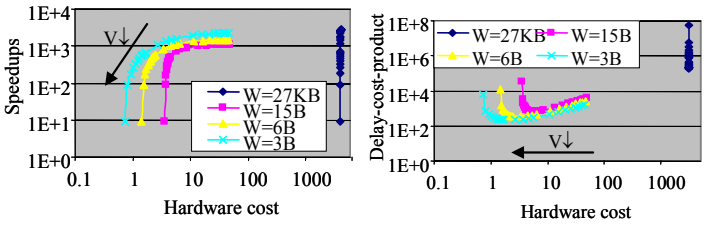
Figure 17 Reducing the width of KB_RAM increases the N_E vector read time

Using the benchmark training file as an example, Figure 17 shows the probability distribution of the number of cycles needed to read an N_E vector when the width of the KB_RAM is 15B, 6B and 3B. It shows that 85% of the N_E reads take only 1 cycle when the width of the KB_RAM is 15 bytes. When the width is 3 bytes (i.e. each row only stores one KB element), only 58% of the N_E reads take 1 cycle. Based on this information, we can derive the average read time of the N_E vector. When the width of the KB_RAM is 15 bytes the average time to read out an N_E vector is 1.2 cycles. If the width reduces to 6 bytes then the time increases to 2.3 cycles. If the width of the KB_RAM is 3 bytes then the average read takes 3.3 cycles. The performance of system is bounded by the latency of the PE that represents the lexicon with the largest number of input knowledge bases. The slowest PE must read and process 20 N_E vectors. Based on the law of large numbers, the expected processing time can be calculated as $20 \times cycle_{avg}$, where $cycle_{avg}$ is the average number of cycles to read an N_E vector. Let $line_{avg}$ denote the average number of RAM lines occupied by an N_E vector, $line_{avg} = cycle_{avg}$. The length of the KB_RAM can be calculated as $line_{avg} \times \Sigma_{j \in T(i)}(\#rows_{KB_{ji}})$, where $\#rows_{KB_{ji}}$ represents the number of non-empty rows in the knowledge base from lexicon j to lexicon i.

Reducing the width of the KB_RAM to W bytes also reduces the number of required adders. Since we only need enough adders to process the data that are read out from the memory, only 2/3*W adders are needed.

To reduce the hardware cost, we can also decrease the number of comparators. Assume that there are V comparators in each PE, the latency of the comparison is approximately estimated as $7 + (2^{\log_2 \max\_row - \log_2 V} - 1)$.

Using the information obtained from the benchmark training file, we can analyze cost-performance tradeoffs for the recall system. Assume that each 1MB RAM has 1 unit area, which is also the area of 100 8-bit adders or comparators. By varying the value of V (i.e. the number of comparators) from 4500 to 1 and varying the value of W (i.e. the capacity of the KB_RAM) we obtain a set of design points. Figure 18 (a) shows where the design points fall in the performance cost space. The X axis gives the hardware cost and the Y axis gives the speed up over sequential processing. The cost is measured as the total area of the hardware. The results show that for the same hardware cost, a narrower KB_RAM is faster. Figure 18 (b) shows the delay-cost-product of the different hardware configurations. For the same cost, a narrower KB_RAM has higher efficiency. A system with too many comparators or too few comparators does not provide the best efficiency. For W=15B~6B, a system with about 60 comparators is the most efficient in terms of the delay and the cost.



(a) Speedups versus cost

(b) Hardware efficiency

Figure 18 Cost performance tradeoff for the recall system.

V. CONCLUSIONS AND FUTURE WORK

In this work we present our research on accelerating confabulation based single sentence completion. Three topics have been covered. They are (1) software optimization, (2) software analysis and (3) architecture exploration. Our analysis shows that the use of appropriate data structures can improve the performance of the software by more than 5000X, and that the cogent confabulation algorithm is an ideal candidate for parallel processing. It also shows that although increasing the number of PEs or the size of memories can increase the performance of training and recall, the relation between hardware cost and performance associated with these variations are not always linear. The details of hardware configuration must be carefully considered to achieve good cost performance tradeoffs. We suggest that this work can be extended to more complex implementations of confabulation systems.

REFERENCES

[1] R. Hecht-Nielsen, "Confabulation Theory: The Mechanism of Thought", Springer, Aug. 2007.

[2] J. R. Anderson, "ACT: A simple theory of complex cognition," *American Psychologist*, Vol. 51, No. 4, pp. 355-365, 1996.

[3] J. Hawkins, S. Blakeslee, "On Intelligence," *Times Books*, 2004.

[4] A. J. Anderson, "A Brain-Like Computer for Cognitive Software Applications: The Ersatz Brain Project," *Proc. of IEEE International Conference on Cognitive Informatics*, Irvine CA, 2005.

[5] H. Moravec, "When Will Computer Hardware Match the Human Brain?" Journal of Evolution and Technology, Vol. 1, 1998.

[6] P. Dubey, "Recognition, Mining and Synthesis Moves Computer to the Era of Tera," *Technology Intel Magazine*, pp. 1-10, February 2005.

[7] G. De Micheli, "Synthesis And Optimization of Digital Circuits", *McGraw-Hill Inc.*, 1994

[8] M. A. Nuno-Maganda, M. Arias-Estrada, C. Torres-Huitzil, "An Efficient Scalable Parallel Hardware Architecture for Multilayer Spiking Neural Networks," *3rd Southern Conference on Programmable Logic*, Feb. 2007.

[9] Q. Wu, Q. Qiu, R. Linderman, D. Burns, M. Moore, D. Fitzgerald, "Architectural Design and Complexity Analysis of Large-Scale Cortical Simulation on a Hybrid Computing Platform," *Proc. of IEEE Symposium on Computational Intelligence in Security and Defense Applications*, 2007.

[10] W. Baek, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun, "Towards Soft Optimization Techniques for Parallel Cognitive Applications," *Proceedings of ACM symposium on Parallel algorithms and architectures*, June 2007.

[11] Codes 'LexMaker.c' and 'TextReader.c' by M. Moore, ITT Systems, AFRL/IFTC, 'cf.c' by Dan Burns, AFRL/IFTC, Rome, NY, and 'conf.cpp' by Q. Qiu, Binghamton Univ.