ACCEPTED:
A. Beg and Y. Chu, "Modeling of Trace- and Block-Based Caches,"
Journal of Circuits, Systems and Computers (JCSC), vol. 16, August 2007.

MODELING OF TRACE- AND BLOCK-BASED CACHES

AZAM BEG[1]

College of Information Technology, United Arab Emirates University,
Al-Ain, Abu-Dhabi, United Arab Emirates (UAE)
abeg@uaeu.ac.ae

YUL CHU

Department of Electrical & Computer Engineering, Mississippi State University,
Mississippi State, MS 39762, USA
chu@ece.msstate.edu

Received (29 April 2006)
Revised (27 February 2007)

Recent cache schemes, such as trace cache, (fixed-sized) block cache, and variable-sized block cache, have helped improve instruction fetch bandwidth beyond the conventional instruction caches. Trace- and block-caches function by capturing the dynamic sequence of instructions. For industry standard benchmarks (e.g., SPEC2000), performance comparison of various configurations of these caches using simulations can take days or even weeks. In this paper, we demonstrate that neural network models can be time-efficient alternatives to the simulations. The models are able to predict the multi-variate and non-linear behavior of trace- and block-caches, in terms of trace miss rate and average trace length. The models can be potentially used in compiler optimization or in pedagogical settings.

Keywords: Basic blocks, trace cache, block cache, variable sized block cache, neural network, cache-modeling, compiler optimization

1. Introduction

The basic storage unit of conventional instruction cache (IC) is a cache line that stores a set of memory-adjacent instructions. IC's tend to possess high latency and low bandwidth [1]. To increase the cache bandwidth sequences of executed instructions are stored in caches such as *trace-cache* (TC), *(fixed-sized) block-cache* (BC) and *variable-sized block cache* (VSBC) [2][3][4]. However, the behavioral study of these caches using simulations can be quite time- and computing resource-intensive. Even the trace-driven simulations with benchmarks such as SPEC2000 can span over days or weeks [5][6]. This paper presents time-efficient cache models as an alternative to simulations. We utilized a set of data collected from SPEC2000 (integer) benchmark simulations on three different trace- and block-based cache schemes and created a multi-variate model based on neural networks (NN's). The NN models (NNM's) are able to predict the trace miss rates and average trace lengths using: (1) cache parameters (e.g., size, associativity, etc.) and (2)

[1] Present address: College of Information Technology, United Arab Emirates University, P O Box 17555, Al-Ain, Abu Dhabi, United Arab Emirates (UAE).

program's characteristics (i.e., counts of basic blocks of different sizes). The NNM predictions match the simulation results with accuracy comparable to analytical models (proposed by other researchers in the past). Our NNM's have potential applications in compiler optimization or pedagogy.

Section 2 of this paper summarizes the introductory topics such as basic (instruction) blocks, conventional caches, trace cache, and block-based caches. This section also includes the basics of NN's and their applications to the field of computer architecture. Section 3 goes into the details of NNM implementation for trace- and block-caches. Details of data collection and processing, NNM training and validation, comparison of simulation and NN modeling speeds are included in this section as well. How the NNM can be used to analyze the cache performance for an arbitrary program is also presented in this section. Section 4 concludes the paper and proposes potential applications and ideas for extending this research.

2. Preliminaries & Previous Research

2.1. *Basic Blocks*

A *basic (instruction) block* is a set of contiguous instructions that contains only a single control instruction, such as a conditional or unconditional jump, or a return. The control instruction is the last instruction of the basic block. The beginning of a basic block is called *block head*, and the end is called *block tail*. Block head is also the destination of a control transfer instruction.

2.2. *Caching*

In a computer system, caching operation is based on the observation that the common programs tend to access contiguous locations in memory (spatial locality) or the same memory locations repetitively (temporal locality). This program behavior results in low latency (how fast the memory contents are available) and higher bandwidth (how much data is readily available) for caches. Effectively, the caches try to approximate the availability of ideally large memory that the programmers expect [1]. One or more levels of cache for data and/or instruction storage can be used between the processor and the main memory; placing fast caches close to the processor reduces memory latency by storing frequently or recently accessed data and instructions [7].

2.3. *Trace Cache*

The basic data unit of conventional instruction cache (IC) is a *cache line* that stores a set of memory-adjacent instructions. The usual cache line lengths are 16 to 64 bytes. IC although simple to implement, tends to be exhibit high latency and low bandwidth. Typically single-ported reads limit IC’s bandwidth to a single basic block, because of a taken jump to a non-adjacent memory location. For better performance, more basic

blocks need to be fetched every cycle, which was made possible by Rotenberg et al's TC and other follow-up schemes (discussed in the next section) [2][3][8].
TC captures instructions in the order they execute. The matching of the starting address of a TC line and the predictions for branches inside the line are the two conditions that cause the delivery of instructions (to the instruction decoder) from TC instead of IC. The block diagram of a TC-based superscalar microarchitecture is shown in Fig. 1 [2][9][10][11].

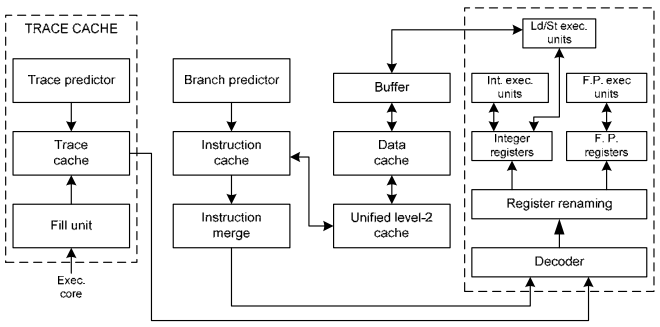

Fig. 1. A superscalar processor with a trace cache (TC)

2.4. *Block Caches*

Black, et al's [3] BC is a variation of TC that includes identification of individual blocks in the stored traces. The block identifiers (pointers) are used to assemble the traces on a trace hit. In BC, two separate cache structures are used, one (called block cache) to store the basic blocks, and the other (called trace table) to store the block pointers. In [3], the block cache structure is replicated 4 times. The underlying assumption that the all basic blocks have fixed length increases the likelihood of block fragmentation. (TC in comparison causes trace-level fragmentation). Black, et al [3] do not compare BC's miss rate and trace length with other cache schemes. The block diagram of Fig. 2 shows a superscalar processor connected to BC. A scheme similar to BC was proposed in a paper by Jourdan, et al [10]. Their scheme called extended block (XB) cache stores the instructions (uops) in reverse sequence, which gives them the ability to extend any existing XB's. The authors report reduced block fragmentation but XB bandwidth is only marginally better than TC [10]. (Due to this minor improvement over TC and significantly complex implementation logic, we will limit our discussion on XB to this section only, and will not use XB for modeling in this paper). Unlike TC, no follow-up research has been reported on either BC or XB, since their introductions.

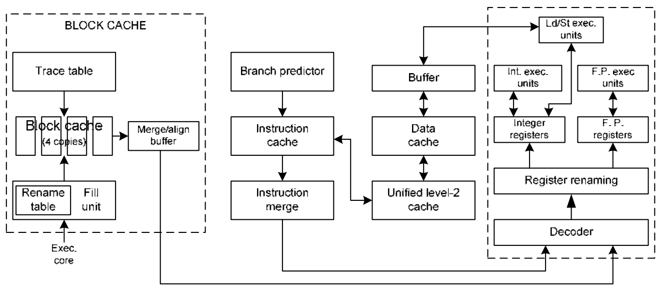

Fig. 2. A superscalar processor with a block cache (BC)

2.5. *Variable-Sized Block Cache*

VSBC was introduced in [4] and addressed some of the short-comings of TC and BC. Block diagram of VSBC when connected to a superscalar processor is shown in Fig. 3. Unlike TC, the basic blocks that may appear in multiple traces are stored only once. The basic blocks are stored in basic block cache (BBC) and the starting and ending addresses of the basic blocks are stored in a separate structure called block pointer cache (BPC). Each line in BPC represents a single trace by storing multiple sets of basic block (start and end) addresses. Unlike BC, storage and retrieval of block sizes of varying lengths are allowed. Combined effects of variability of block sizes and set-associativity in BBC manifest in lower trace miss rate than TC and BC; VSBC's average number of instructions stored per trace is also higher than both TC and BC. Cache storage in VSBC is more efficient than BC because the latter replicates cache structures. TC uses only the beginning address of a trace for matching. Blocks other than the beginning block are not identifiable, so even if the required instructions are present in the trace, the trace is declared a 'miss' and a new trace build is initiated. VSBC avoids this unnecessary switching from trace utilization mode to build mode, by allowing hits even on intra-trace blocks; this helps improve trace miss rate.

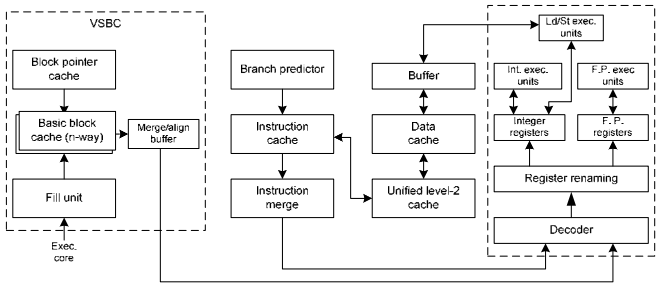

Fig. 3. A superscalar processor with variable-sized block cache (VSBC)

2.6. *Neural Networks*

Neural networks (NN's) mimic the ability of a human brain to find patterns and uncover hidden relationships in data. NN's can be more effective than statistical techniques for organizing data and predicting results, and can be quite efficient in modeling non-linear systems [12][13][14].
A neural network (NN) is defined as a computational system comprising of simple but highly interconnected processing elements (PE's) (or 'neurons') (Fig. 4 and Fig. 5) [15]. PE's are neural network equivalents of biological neurons. Unlike conventional computers that process instruction and data stored in the memory in a sequential manner, the NN's produce outputs based on a weighted sum of all inputs in a parallel fashion [12].

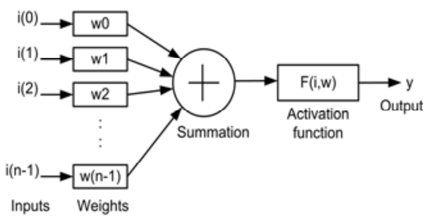

Fig. 4. Processing element (PE), the building block of a neural network. The inputs (*i(0)..i(n-1)*) are scaled (multiplied) with weights (*w(0)..w(n-1)*) and summed up before being passed through a (linear or non-linear) activation function.

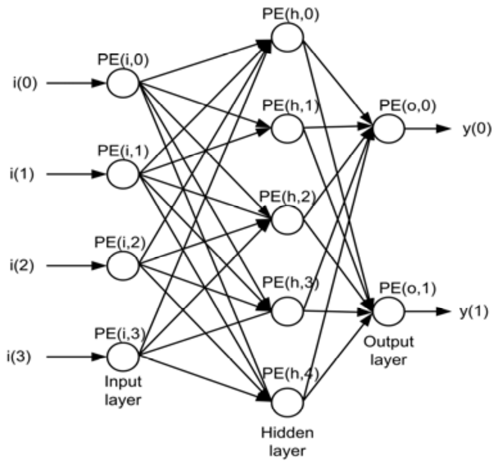

Fig. 5. Topology of a simple 3-layer feed-forward neural network. The NN has 4 inputs (*i(0)..(i(3)*) feeding into 4 input neurons (*PE(i,0)..PE(i,3)*) and 2 outputs (*y(0)..y(1)*) driven by (*PE(o,0)..PE(o,1)*). There is a single hidden layer composed of 5 neurons (*PE(h,0)..PE(h,4)*).

NN's use different types of learning (or training) mechanisms, the most common of them being supervised learning. In this method of learning, a set of inputs is provided to the

NN and its output is compared with the desired output. The difference of actual and desired outputs is used to adjust the connection weights to different PE's in the network. The process of adjusting weights is repeated until the output falls within an acceptable range [12].

For effective NN training, the data may need to be transformed (by reformatting and/or scaling). Also, the data is divided into training and validation sets. An NN is trained only with the training set whereas validation set is used to verify that the NN produces desirable outputs with 'unseen' inputs. If the validation phase produces large deviations, the training set or the network structure needs to be re-examined; re-training is required in this case [12].

2.7. *Neural Network Applications in the Realm of Computer Architecture*

When compared with actual implementation or simulation, the models are usually faster for studying the design or operation of a system [16]. As stated earlier, NNM's are made up of a set of weights that are applied to the model inputs to calculate the outputs. While being robust, NNM's can also be good alternatives to (1) analytical models and (2) lookup methods that require storage of all data points in given data space [17].

Several applications of NNM's to application to the field of computer architecture have been reported in the past years. A few examples of such research are given below.

Khalid [18][19] presented an NNM-based cache replacement scheme that performed better than LRU scheme in case a program has more spatial locality than temporal locality. His research reportedly would save chip area by using a single level of cache instead of multiple levels.

Calder, et al [20] used a NN to make static branch predictions. Their technique involved using a collection of programs to model the behavior of branches in a new program. They used NN's and decision trees to extract the static properties of different branches in order to predict a branch. Their NNM eliminated the need for rule-based branch prediction mechanisms. Calder, et al [20] also reported that their branch prediction technique resulted in 20% miss rate vs. 25% miss rate of then-available heuristics-based methods.

Jimenez and Lin [21] came up with a NNM for branch prediction. The size of their NN-based branch predictor increases linearly as compared to a conventional branch predictor whose hardware enlarges exponentially when the branch histories are extended. Their global and global-and-local branch predictors not only showed reduced miss rates but also resulted in better IPC (instructions per cycle) than traditional (two-bit, McFarling hybrid) branch predictors.

Gomez et al [22] proposed a NN method for administering the resources in chip multi-processors. Their technique of dynamically assigning the L2 cache banks to a set of processor cores resulted in noticeably better performance than static partitioning method.

Although many analytical models for conventional caches [24][25][26][27][28] have been proposed, no research so far seems to have covered the subject of an integrated mathematical or NN modeling of trace- and block-caches. So we consider our research to be the first attempt in this direction. In our paper, the primary reason for using NNM's for cache modeling is the ease and effectiveness of NN's in modeling the non-linear and multi-variate systems.

3. The Neural Network Model for Trace and Block Caches

3.1. *Cache Performance Criteria*

In order to compare the performance of TC, BC, and VSBC, two metrics, namely, trace miss rate and average trace length were used in [4]; these metrics were considered to be among the most appropriate in the context of trace- and block-based caches. The same two metrics were used in constructing our NNM's. Trace miss rate is the percentage of references when a requested trace was not found in the cache. Smaller value of trace miss rate represents a lower average latency for cache data fetching. The larger the average trace length, the more the number of instructions fetch-able per cycle.

3.2. *Data collection*

For the NNM's in this paper, the training and validation sets were obtained by running different configurations of trace-driven cache simulators: Sim-TC for TC, Sim-BC for BC, and Sim-VSBC for VSBC [4]. (Note that our work in [4] was limited to simulations; the modeling efforts are only being presented this paper). Each simulator ran ten SPECint2000 benchmark programs [29] (Table 1) on all the cache configurations listed in Table 2. For each of these simulations, we acquired two values: (1) trace miss rate, and (2) average trace length. We used the basic block counts and sizes (Fig. 6) as the parameters that would represent a benchmark; the basic block statistics were gathered using SimpleScalar tool [30][31].

Table 1. Benchmarks Used for TC, BC, and VSBC Simulations

Benchmark	Description	Input Data Set
bzip	Compression	input.random
crafty	Game playing: chess	crafty.in
gap	Group theory, interpreter	test.in
gcc	C language compiler	cccp.i
gzip	Compression	input.compressed
mcf	Combinatorial optimization	inp.in
parser	Word processing	test.in
perlbmk	PERL language	test.pl, test.in
vortex	Object-oriented database	lendian.raw
vpr	FPGA placement & routing	net.in, arch.in

Table 2. Simulation Parameters for TC, BC, and VSBC

Parameter	TC	BC	VSBC
Max number of traces	64, 128, 256, 512, 1024, 2048	64, 128, 256, 512, 1024, 2048 (lines in trace table)	64, 128, 256, 512, 1024, 2048 (NBPC)
Number of ways in TC/BPC	1	1	1
Cache capacity (KB)	1, 2, 4, 8, 16, 32, 64, 128, 256	1, 2, 4, 8, 16, 32, 64, 128, 256 (block cache)	1, 2, 4, 8, 16, 32, 64, 128, 256 (BBC)
TC/BBC associativity (ways)	1, 2, 4, 8, 16	1, 2, 4, 8, 16 (block cache)	1, 2, 4, 8, 16 (WBBC)
Max basic blocks per trace	4	4	4
Max possible number of instructions per trace	16	16	Not limited
Entries in branch history table	1024	1024	1024

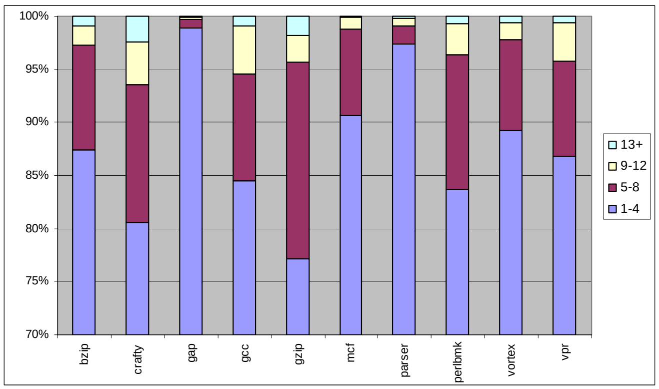

Fig. 6. Block length distribution of different SPECint2000 benchmarks; runs were limited to 100m instructions.

3.3. *Neural Network Model Definition*

The NNM's in this paper predict (output) trace miss rate and average trace length for TC, BC, and VSBC. Inputs to the NNM's are cache configuration and program characteristics (block counts) as shown in Table 3. Note that cache type is a *symbol*, rather than a value, so 3 discrete inputs #4, 5, and 6 are used to represent the cache types [12]: TC = {1, 0, 0}; BC = {0, 1, 0}; VSBC = {0, 0, 1}.

Our initial attempts at NN modeling used a single value for the average block size (for the complete run). But, we discovered that the block size averages among the benchmarks were not distinct enough to properly represent the benchmarks for the purposes of NN-training. So, for each benchmark, we used more than one value of block count as shown in Fig. 6. Two of the several NNM configurations, we experimented with, are shown in Table 3. We chose Configuration 2 for the final NNM's due to their better training performance.

Table 3. NNM Configurations - Input and output neuron definitions. The difference between the two configurations is the representation of a program using block counts, i.e., inputs # 7, 8, 9, and 10.

Input/ Output Neurons	Configuration 1	Configuration 2
Output	Trace miss rate/average trace length	Trace miss rate/average trace length
Input 1	Traces in TC -or- Lines in BC's Trace Table -or- Lines in VSBC's BPC (NBPC)	Traces in TC -or- Lines in BC's Trace Table -or- Lines in VSBC's BPC (NBPC)
Input 2	Cache capacity (KB): TC size -or- Block cache capacity in BC -or- BBC size in VSBC	Cache capacity (KB): TC size -or- Block cache capacity in BC -or- BBC size in VSBC
Input 3	TC ways -or- Block cache ways in BC -or- BBC ways in VSBC (WBBC)	TC ways -or- Block cache ways in BC -or- BBC ways in VSBC (WBBC)
Input 4	Cache type is TC	Cache type is TC
Input 5	Cache type is BC	Cache type is BC
Input 6	Cache type is VSBC	Cache type is VSBC
Input 7	% of blocks with 1 to 4 instructions	% of blocks with 1 to 4 instructions
Input 8	% of blocks with 5 or more instructions	% of blocks with 5 to 8 instructions
Input 9		% of blocks with 9 to 12 instructions
Input 10		% of blocks with 13 or more instructions

3.4. *Data Pre-Processing*

Pre-processing the training and validations sets can take a considerable amount of resources for a practical and reliably functioning NNM [14][32]. In our research, the first data pre-processing step was to apply Z-score, a statistical technique of specifying the degree of deviation of a data value from the mean. Z-score is calculated by this formula [33][34]:

$$Z = \frac{(x - \bar{x})}{\sigma} \tag{1}$$

where x is the individual value, $\bar{x}$ is the sample mean, and σ is the sample standard deviation.
As a 2nd step of pre-processing, we normalized the training set to the range [0, 1]; normalization was done to ensure equitable distribution of importance among inputs. In other words, the larger absolute values of an input should not have more influence than the inputs with smaller magnitudes [35]. Similarly, we also normalized the outputs to the [0, 1] range [36]. For n samples, the [0, 1] normalization was a 2-step process:

$$x'_i = x_i - x_{\min}, \quad i = 0 \,..\, n-1 \qquad (2)$$

$$x''_i = x'_i / x'_{\max}, \quad i = 0 \,..\, n-1 \qquad (3)$$

For the cache-size input values that are multiplicative in nature (i.e., 1K, 2K, 4K …), we used $\log_2$ transformation prior to normalization of equations 2 and 3 [35].

3.5. *Neural Network Training Results and Analysis*

We used an NN-modeling software package called Brain Maker (version 3.75 based on MS-Windows) [37] to create and test our NNM's. Brain Maker's back-propagation NN's were fully connected, meaning all inputs were connected to all hidden neurons, and all hidden neurons were connected to the outputs. The activation function for the hidden and output layers was a sigmoid function.

We acquired a total of 150 data sets (also called facts/training facts) during our simulations of TC, BC, and VSBC. (Simulation details have been explained in Section 3.2.) 120 data sets were used for training set, while the remaining 30 were used for validation. We stopped an NN training session, when the sooner of these two conditions was met:

- Epoch count reached 30000. Rationale: Most of our properly NNM's converging so much before this count.
- 90% of the facts were learnt with less than 10% mean squared error (MSE). Rationale: 90% fact limit keeps NNM topology small and achieve generalizations instead of 'rote' learning [13]. The allowance of 10% MSE is comparable to other cache models[2].

A general rule is that as the number of hidden layers increases, the prediction performance goes up, but only up to a certain point, after which the NNM performance starts to deteriorate [32]. To find the optimum topologies for our NNM's, we experimented with up to 3 hidden layers; each layer consisted a different number of neurons. Brain Maker assigns random values to all the weights at the beginning of every training session [37]. Details of some of our NNM's experiments are listed in Table 4 (trace miss rate) and Table 5 (average trace length). The miss rate NNM was able to learn 91% of the facts and the average trace length NNM learnt 82% of the facts. (Both NNM's were allowed a maximum of 10% MSE). Similarity in the values of block sizes in the training set seems to be the reason for difficulty in training the average trace length NNM with any higher accuracy.

[2] (1) 15% error in Harper, et al's [38] cache analytical model for miss ratio predictions, (2) 7% error in Hossain's [6] TC mathematical model for instruction fetch rate. (4) 14% error in Hossain, et al's [5] TC model for average trace length, and (5) up to 37% error in Agarwal, et al's [23] mathematical model for cache misses.

Table 4. Training performance of the trace miss rate NNM. Optimum results were achieved with a 4-Layer (10-5-5-1) NNM (shown in bold)*

NNM Size (Number of neurons)					Stop training when		
Input layer	Hidden Layer 1	Hidden layer 2	Hidden Layer 3	Output	Epochs	Learnt 90%+ facts	Training Accuracy
10	10	-	-	1	30000	no	71%
10	20	-	-	1	30000	no	69%
10	10	5	-	1	14500	yes	91%
10	7	5	-	1	30000	no	71%
10	**5**	**5**	**-**	**1**	**16471**	**yes**	**91%**
10	10	10	-	1	2900	yes	91%
10	15	10	-	1	5167	yes	91%
10	10	10	5	1	3008	Yes	91%

* Brain Maker training parameters: Training tolerance = 0.1; testing tolerance = 0.1; learning rate (initial value) = 0.1; learning rate adjustment type = exponential. (See [37] for details).

Table 5. Training performance of the average trace length NNM. Optimum results were achieved with a 4-Layer (10-15-10-1) NNM (shown in bold)*

NNM Size (Number of Neurons)					Stop Training When		
Input layer	Hidden Layer 1	Hidden layer 2	Hidden Layer 3	Output	Epochs	Learnt 90%+ facts	Training Accuracy
10	10	-	-	1	30000	no	78%
10	20	-	-	1	30000	no	35%
10	10	5	-	1	30000	no	68%
10	7	5	-	1	30000	no	37%
10	5	5	-	1	30000	no	37%
10	10	10	-	1	30000	no	76%
10	**15**	**10**	**-**	**1**	**30000**	**no**	**82%**
10	10	10	5	1	30000	no	62%

* Brain Maker training parameters: Training tolerance = 0.1; testing tolerance = 0.1; learning rate (initial value) = 0.1; learning rate adjustment type = exponential. (See [37] for details).

Fig. 7 shows the comparison of simulated ('actual') and NNM's predicted values for VSBC trace miss rates. Fig. 8 compares the simulations and predictions for average trace lengths. As to the horizontal axis labels, in these figures, '*bzip_512*' refers to trace miss rate/average trace length of '*bzip*' benchmark when N_{BBC} = 512 lines, and so on. We measured the simulation-prediction average error to be 10.9% for miss rate and 13.5% for average trace length. These errors are within the ranges of other cache models. (See footnote on page 10.)

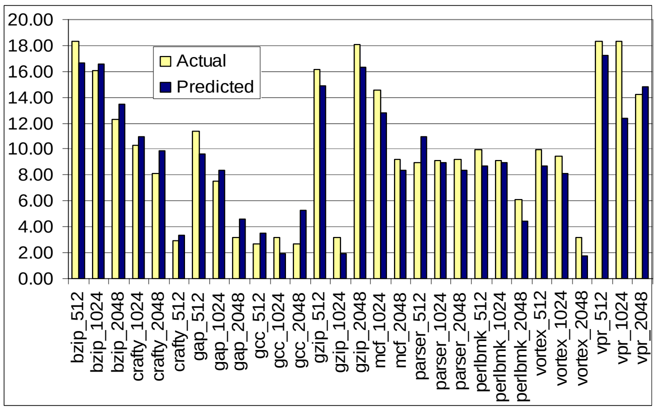

Fig. 7. Effect of changing VSBC's N_{BBC} on trace miss rate. N_{BBC} = {512, 1024, 2048}, W_{BBC} = 1, N_{BPC} = 256. Horizontal axis shows benchmark name and N_{BBC} and vertical axis shows the miss rates.

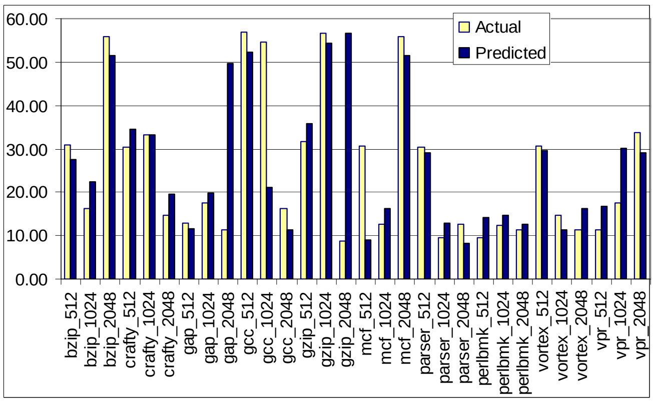

Fig. 8. Effect of changing VSBC's NBBC on average trace length. N_{BBC} = {512, 1024, 2048}, W_{BBC} = 1, N_{BPC} = 256. Horizontal axis shows the benchmark name and N_{BBC} and vertical axis shows the average trace length.

3.6. *Using the Neural Network Models*

In this section, we show a few examples of the insight one can gain by using the NNM's. We arbitrarily picked a set of values of the block sizes to examine VSBC's behavior: {0.80, 0.17, 0.03, 0.02} which means that 80% of blocks contained 1 to 4 instructions, 17% blocks contained 5 to 8 instructions, 3% blocks contained 9 to 12 instructions, and

2% blocks contained 13 or more instructions. Note that these block sizes were different from any of the benchmarks' block sizes used to train or validate the NNM's.
As expected, the results in Fig. 9 show us that increase in BPC size reduces the miss rate. The average trace lengths, however, vary slightly but do not show a trend (Fig. 10). Currently, even a single block hit (partial hit) is considered a trace hit. Changing the definition of partial hits to two or more blocks may result in higher averages of trace lengths; although this redefinition of partial hits may reduce the trace miss rate.

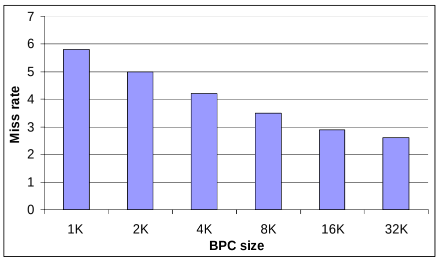

Fig. 9. Effect of varying VSBC cache (BPC) size on miss rate: A drop in miss rate happens with increase in BPC capacity.

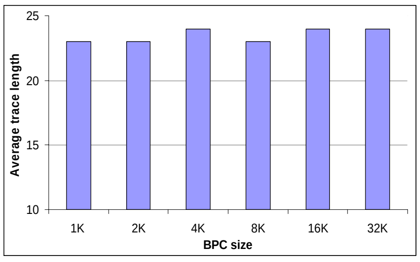

Fig. 10. Effect of varying VSBC cache (BPC) size on average trace length. The average trace length vary but do not show a trend.

We see that increasing BBC associativity from 1 to 2 has the largest miss rate improvement, but the gains flattens out with associativities of 4 and higher (Fig. 11); the trend resembles the familiar cache-associativity behavior. The trace lengths do not vary much but peak out with a 4-way BBC (Fig. 12). The reason for little change in trace lengths may be that the storage of a basic block is restricted to a single BBC-way, so the higher associativity does not change appreciably the average trace length.

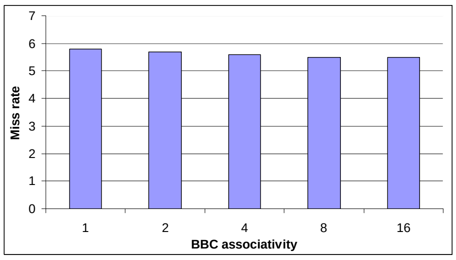

Fig. 11. Effect of varying VSBC-BBC associativity on miss rate. After an initial drop in miss rate, it flattens out with increase in associativity.

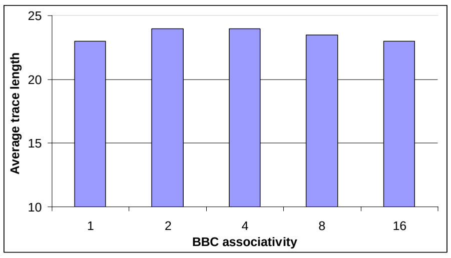

Fig. 12. Effect of varying VSBC-BBC associativity. The trace lengths vary slightly with change in BBC-associativity.

Additionally, we used the NNM's to compare the performance of different configurations of TC, BC, and VSBC. Here we use the same input program as the previous examples, i.e., with the block percentages of {0.80, 0.17, 0.03, 0.02}. The predicted values of trace miss rate and average trace lengths are shown in Fig. 13 and Fig. 14. We observe that miss rates improve as cache size increases, but the improvement rate tends to flatten out after 8K cache size. VSBC offers better miss rates than TC and BC for all cache sizes. Trace lengths for a given cache scheme remain relatively stable, while VSBC maintains its lead over both TC and BC.

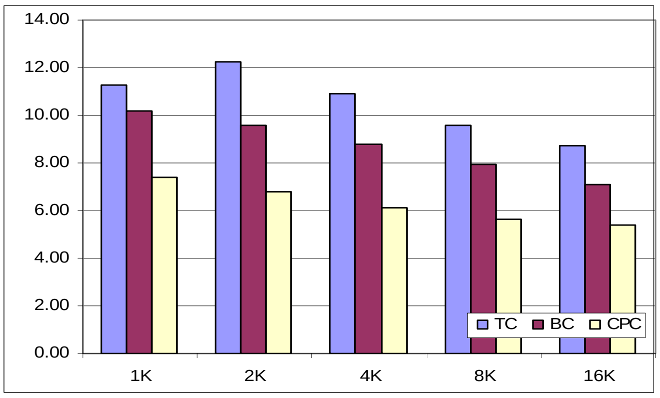

Fig. 13. For a program with arbitrarily chosen 'block size distribution' of {0.80, 0.17, 0.03, 0.02}, miss-rate NNM was used to predict the values for TC, BC, and VSBC. The horizontal axis shows cache size and the vertical axis represents miss rate percentages.

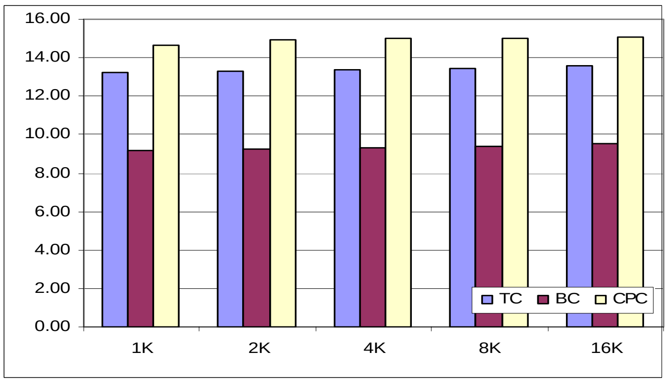

Fig. 14. For a program with arbitrary chosen 'block size distribution' of {0.80, 0.17, 0.03, 0.02}, trace-length NNM was used to predict the values for TC, BC, and VSBC. The horizontal axis shows cache size and the vertical axis represents the trace length in terms of number of instructions.

3.7. *Neural Network Model Speed*

As anticipated, running the NNM's takes much less time than the trace-driven simulations. These time comparisons are given in Table 6. The NNM's generate predictions in an average of 0.07 second whereas the average simulation times are 35342 seconds. Simulation times range from 13830 seconds (row 1) to 61464 seconds (row 2). (A Windows-XP based, 2.4 GHz Pentium-4 machine has been used in these experiments).

Table 6. Comparison of simulation time and NNM running times. The NNM generate prediction in an average of 0.07 second whereas the average simulation times are 35342 seconds. Minimum simulation time is 13830 seconds (row 1) and the maximum is 61464 seconds (row 2).

N_{BPC}	N_{BBC}	W_{BBC}	Simulation Time (seconds)	NNM Running Time (seconds)
64	512	1	13830	0.07
512	2048	4	61464	0.07

4. Conclusions & Further Research

In this paper, we presented a new and unified model for trace- or block-based traces using NNM's. Until the time of this writing, no such models have been reported by other researchers. Although relatively interpolative in nature, once the NNM's have been developed, they can be used to conduct further experiments with different types of inputs, in a fraction of the time what a simulator would take (i.e., 0.07 second vs. 35242 seconds.) Our miss rate NNM makes predictions with 10.9% accuracy and average trace length NNM produces outputs with 13.5% accuracy. We also looked at some examples of NNM usage with some arbitrary input values that were 'shown' to the NNM neither during training nor validation.

Two potential applications of this paper's NNM's (which are a subject of a future study) are given below:

- Time-efficient estimation of how the performance of different cache configurations relates to the compiler optimization techniques that affect the block sizes, such as branch elimination, conversion of control dependence into data dependence, etc.
- Quick demonstration, especially for pedagogical purposes, of the effect of code block sizing on the performance of different caches.

Two other ideas for extending this paper's research are:

- The training set for cache NNM's could be expanded to include a wider range of input parameters, such as number of threads (in a multi-threading environment), floating-point benchmarks, etc.
- If the NNM training data were obtained from full-processor simulations instead of just the caches, the NNM's could provide insight into processor specific parameters such as cycles per instruction (CPI), execution time, etc.

References

1. J. Hennessy, and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., (Morgan Kaufman Publishers, California, 2003).
2. E. Rotenberg, S. Bennett, and J. E. Smith, A Trace Cache Microarchitecture and Evaluation, *IEEE Trans. Comp.* **48** (1999) 111 – 120.

3. B. Black, Rychlick, and J. Shen, The Block-Based Trace Cache, *Proc. Int. Symp. Comp. Arch.*, 1999 (Atlanta, Georgia, 1999), pp. 196 – 207.
4. A. Beg and Y. Chu, Improved Instruction Fetching with a New Block-Based Cache Scheme, *Peoc. Int. Symp. Signals, Circuits & Systems*, 2005 (Iasi, Romania, 2005), pp. 765-768.
5. A. Hossain, D. J. Pease, J. S. Burns and N. Parveen, Trace Cache Performance Parameters, *Proc. IEEE Int. Conf. Computer Design,* 2002 (Freiburg, Germany 2002), pp. 348-355.
6. A. Hossain, Trace Cache in Simultaneous Multi-threading, PhD dissertation, Syracuse University, USA, 2002.
7. T. Shanley and D. Anderson, ISA System Architecture, (Addison-Wesley Publishing Company, Boston, Massachusetts, 1995).
8. T. Conte, K. Menezes, P. Mills and B. Patel, Optimization of instruction fetch mechanisms for high issue rates, *Proc. Int. Symp. Comp. Arch.,* 1995 (Santa Margherita Ligure, Italy, 1995), pp. 333-344.
9. J. Gummaraju and M. Franklin, "Branch prediction in multi-threaded processors," *Proc. Int. Conf. Parallel Arch. and Compilation Techniques,* 2000 (Philadelphia, Pennsylvania, 2000), pp. 179-188.
10. S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz and R. Ronen, eXtended Block Cache, *Proc. Int. Symp.High-Performance Comp. Arch.* 2000 (Toulouse, France 2000), pp. 61-70.
11. S. Patel, D. Friendly and N. Y. Patt, Evaluation of Design Option for the Trace Cache Fetch Mechanism, *IEEE Trans. on Computers,* 48 (1999) 193 –204,
12. M. Caudill, *AI Expert: Neural Network Primer*, (Miller Freeman Publications, San Francisco California, 1990).
13. R. E. Uhrig, Introduction to Artificial Neural Networks, *Proc. IEEE Int. Conf. Industrial Electronics, Control and Instrumentation*, 1995 (Nagoya, Japan, 1995), pp. 33-37.
14. K. Yale, Preparing the right data for training neural networks, *IEEE Spectrum* **34** (1997) 64-66.
15. G. Stegmayer and O. Chiotti, Identification of Frequency-Domain Volterra Model Using Neural Networks, *Proc. Int. Conf. Artificial Neural Networks,* 2005 (Warsaw, Poland, 2005), pp. 465-471.
16. A. J. Smith, The Need for Measured Data in Computer System Performance System Analysis or Garbage in, Garbage out, *Proc. Int. Comp. Software and Applications Conf.*, 1994 (Los Alamitos, California, 1994), pp. 426-431.
17. T. W. Simpson, J. Peplinski, P. N. Koch and J. K. Allen, On the Use of Statistics in Design and the Implications for Deterministic Computer Experiments, *Proc. ASME Des. Eng. Tech. Conf.* 1997 (Sacramento, California, 1997), pp. -.
18. H. Khalid, A Neural Network-Based Replacement Strategy for High Performance Computer Architectures, PhD dissertation, City University of New York, The City College, New York, 1996.
19. H. Khalid and M. S. Obaidat, *KORA: A New Cache Replacement Scheme*, Computers & Electrical Engineering **(**Ingenta Connect, Bath, UK, 2000), pp. 187-206.
20. B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer and B. Zorn, Evidence-Based Static Branch Prediction Using Machine Learning, *ACM Trans. Prog. Lang. and Systems* **19** 1997, pp. 188 – 222.
21. D. A. Jimenez and C. Lin, Dynamic Branch Prediction With Perceptrons, *Proc. Int. Symp. High-Perf. Comp. Arch.* 2001 (Nuevo Leone, Mexico, 2001), pp. 197 – 206.

22. F. Gomez. D. Burger, and R, Mikkulainen, A Neuroevolution Method for Dynamic Resource Allocation on a Chip Multiprocessor, *Proc. of the Int. Joint Conf. Neural Networks* 2001 (Washington, District of Columbia, 2001), pp. 2355 – 2360.
23. A. Agarwal, M. Horowitz, and J. Hennessy, An Analytical Cache Model, *ACM Trans. on Comp. Sys.*, vol. 7, 1989, pp. 184 – 215.
24. S. Ghosh, M. Martonosi, and S. Malik, Cache Miss Equations: An Analytical Representation of Cache Misses, *Proc. ACM Int. Conf. Supercomputing* 1997 (Vienna, Austria, 1997), pp. 317 – 324.
25. B. B. Fraguela, R. Doallo, and E. L. Zapata, Automatic Analytical Modeling for the Estimation of Cache Misses, *Proc. Int. Conf. Parallel Arch. and Compilation Techniques* 1999 (Newport Beach, California, 1999), pp. -.
26. G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Application to Cache Partitioning, *Proc. ACM Int. Conf. Supercomputing* 2001 (Sorrento, Italy, 2001), pp. -.
27. X. Vera and J. Xue, Let's Study Whole Program Cache Behavior Analytically, *Proc. High-Perf. Comp. Arch.* 2002 (Boston, Massachusetts, 2002), pp. 175-186.
28. D. Chandra, F. Guo, S. Kim, and Y. Solihin, Predicting Inter-Thread cache Contention on a Chip Multiprocessor Architecture, *Proc. High-Perf. Comp. Arch.* 2005 (San Francisco, California, 2005), pp. 340 – 351.
29. CINT2000 (Integer Component of SPEC CPU2000) (2006), http://www.spec.org/cpu2000/ CINT2000/
30. D. Burger and T. Austin, The SimpleScalar Tool Set, Version 2.0, University of Wisconsin-Madison Computer Sciences Department Technical Report #1242, June 1997.
31. SimpleScalar Documentation (2006), http://www.simplescalar.com
32. J. Lawrence, *Introduction to Neural Networks – Design, Theory and Applications*, (California Scientific Software Press, Nevada City, California, 1994).
33. What's a Z-Score and Why Use it in Usability Testing? (2006), http://www.measuring usability.com /z.htm
34. M. Triola, *Elementary Statistics*, 6th ed., (Addison-Wesley Publishing Co, Boston, Massachusetts, 1994).
35. T. Masters, *Signal and Image Processing with Neural Networks*, (John Wiley & Sons, Inc. Hoboken, New Jersey, 1994).
36. G. Wolfe and R. Vemuri, Extraction and Use of Neural Network Models in Automated Synthesis of Operational Amplifiers, *IEEE Trans. Computer-Aided Design of Integ. Circ. and Sys.*, **22** (2003)
37. *Brain Maker – User's Guide and Reference Manual*, 7th ed., (California Scientific Software Press, Nevada City, California, 1998).
38. J. S. Harper, D. J. Kerbyson and G. R. Nudd, Analytical Modeling of Set-Associative Cache Behavior, *IEEE Trans. on Computers*, **48** (1999) 1009-1024.