A Perceptron-Based Task Predictor for Multi-Core Processor Architectures

Jongbok Lee

Dept. of Information and Communications Engineering
Hansung University
Seoul, Republic of Korea
Email: jblee@hansung.ac.kr

***Abstract*—In order to increase the performance of multi-core system processors, the task predictor which speculatively fetches and allocates tasks to each core should be highly accurate. In this paper, a perceptron-based task predictor is proposed for the multi-core processor architectures. Using SPEC 2000 benchmarks as input, the trace-driven simulation has been performed for the dual-core to octa-core processors employing perceptron-based task predictor extensively. Its performance is compared with the architecture which utilizes the conventional two-level adaptive task predictor.**

Keywords–multi-core processor, perceptron

I. INTRODUCTION

Currently, multi-core processors are widely used for the high performance of the computer system, such as smart phones, tablet PCs, notebook computers, and desk top computers, etc [1]–[6]. By utilizing a task predictor, a program is partitioned into speculative multiple tasks which are assigned to the processing core units. Hence, the task predictor should be very accurate in order to effectively take advantage of a multi-core processor architecture. Recently, neural networks such as perceptrons are widely used in the digital systems, which can take advantage of learning. In this paper, a perceptron-based task predictor for multi-core processor is proposed. The SPEC2000 integer benchmark programs are used for estimating the performance of multi-core processors using perceptrons. The result is compared with the performance of the multi-core processor with the conventional scheme.

This paper is organized as follows. In the Section 2, the perceptron-based task predictor will be discussed. The simulation environment will be described in the Section 3. In the Section 4, the simulation results will be analyzed. Finally, the Section 5 concludes our paper.

II. RELATED STUDIES

Perceptron is the neural network capable of learning by producing outputs combined with the inputs and the associated weight values. In the past studies, it has been adopted for predicting branches in the computer systems [7]–[9]. Figure 1 describes the graphic model of a perceptron. A perceptron is represented by weight vectors, which are composed of positive or negative integers. The output is the dot product of the weight vector $w_{0..N}$ and the input vector $x_{0..N.}$ The first element x_0 is always set to 1 to serve as a bias input.

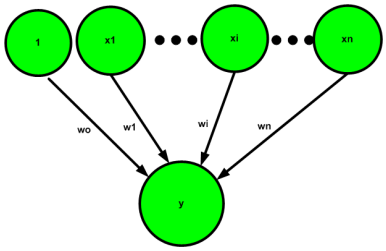


Figure 1. The perceptron.

$$y = w_0 + \sum_{i=1}^{n} x_i w_i \quad (1)$$

Therefore, before adapting to the previous branch results, the biased weight w_0 always enables the perceptron to be biased in the initial stage. The output of a perceptron is represented as (1). The input to a perceptron is bipolar, which means that the branch is not taken if x_i is -1, and taken if x_i is 1. When the output is negative, the branch is predicted as not taken; When the output is positive, it is predicted as taken. When a branch is met, the branch address is used to generate an index between 0 and N-1 to access the perceptron table. After obtaining the weight vector $P_{0..N}$ by fetching the i_{th} perceptron, the dot product of the weight vector and the global history register is generated to output y. The direction of the next branch is predicted upon the sign of the output. When the actual direction of the branch is available, the result is used to update the weight value of the vector P. Then, the vector P is recorded to the i_{th} entry of the perceptron table.

After the perceptron output y has been computed, the following algorithm is used to train the perceptron. Let t be -1 if the branch was not taken, or 1 if it was taken, and let be the threshold, a parameter to the training algorithm used to decide when enough training has been done. Since t and x_i are always either -1 or 1, this algorithm increments the i_{th} weight when the branch outcome agrees with x_i, and decrements when it disagrees.

III. THE PERCEPTRON-BASED TASK PREDICTOR

The task prediction of multi-core processors using perceptrons can be implemented in the similar mechanism as the branches are predicted. Figure 3 illustrates the mechanism

 ISBN: 978-1-61208-466-4

$$
\begin{aligned}
&\textit{if } sign(y_{out}) != t \textit{ or } |y_{out}| <= \theta \\
&\quad \textit{then} \\
&\quad\quad \textit{for } i = 0 \textit{ to } n \textit{ do} \\
&\quad\quad\quad w_i = w_i + tx_i \\
&\quad\quad \textit{end for} \\
&\quad \textit{end if}
\end{aligned}
$$

Figure 2. The perceptron algorithm

of the perceptron-based predictor. The predictor records the finite length of task history results to the task history register and accesses the weight vector table to make a prediction. A temporary task history register is utilized, and at the beginning of each multiple task prediction, the contents of the task history register is transferred to the temporary task history register. In order to predict multiple tasks, the task's starting address

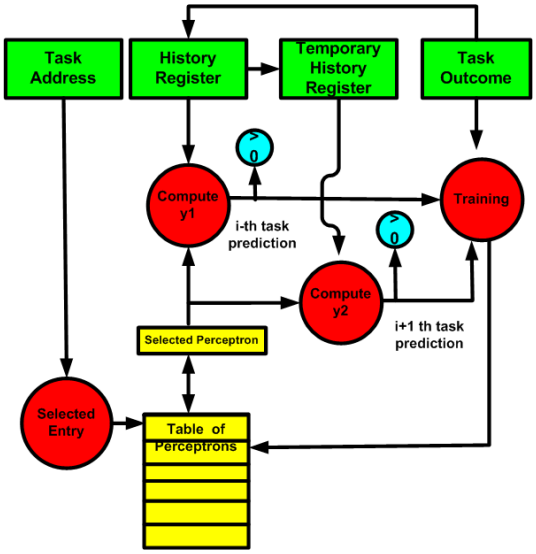


Figure 3. The perceptron-based task predictor

is hashed by the k-bits of history register which is used to index the weight vector table and to predict the i_{th} task. The $i + 1_{th}$ task is predicted on the assumption that the first task prediction is correct. For this purpose, the rightmost 1-bit of the temporary task history register is updated and multiplied to an indexed weight vector to make the $i+1_{th}$ task prediction. In this way, two tasks can be predicted per cycle. Later, when the task outcomes are known, the task history register is updated according to the results. Similarly, to predict the $i + 2_{th}$ task, the rightmost 2 bits of the temporary task history register is updated based on the first and the second task predictions.

IV. The Simulation Environment

A. The multi-core processor architecture

Figure 4 shows the multi-core processor with N cores. Each core is an out-of-order superscalar processor which can execute instructions in a task [10]. In addition, it has a L1 instruction

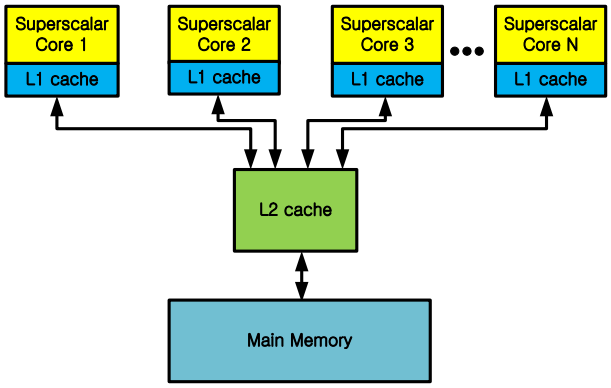


Figure 4. The multicore processor

cache and a L1 data cache. For the cache coherency of the L1 data cache, MESI protocol is utilized. If the data in the L1 data cache associated with a core is over-written by another core, it is invalidated. The L2 cache is shared among the cores, which is connected with the main memory.

The superscalar processor core is allocated with tasks which consist of a number of instructions. The fetched instructions in the task are decoded, renamed, executed, and written back. When all the instructions in the task are retired and becomes empty, new instructions of task are fetched. If the task is mispredicted, the fetch is aborted, and all the remained instructions in the task are squashed. Since the instructions are renamed, the instructions can be issued and executed out-of-order as long as there is no true-dependency. Although the instructions can be retired out-of-order, the instructions are inserted into the reorder buffer and committed in-order as to preserve the original program order.

The detailed architecture configurations and cache parameters for each core are listed in Table I. The number of simulated cores are 1, 2, 4, and 8. Each core is assigned with the maximum of two tasks respectively. Since the small task size cannot take the benefit of the instruction level parallelism, the task sizes are set to 4, 8, and 16. The functional unit of each

TABLE I. ARCHITECTURE CONFIGURATION FOR EACH CORE.

Item		Value
number of cores		1,2,4,8
number of tasks per core		1,2
task length		4,8,16
fetch,issue,retire rate		2,4,8
functional unit	integer ALU	2,4,8
	load/store	1,2,4
L1-instruction cache		64 KB, 2-way set assoc., 16 B block, 10 cycles miss penalty
L1-data cache		64 KB, 2-way set assoc., 32 B block 10 cycles miss penalty
task address cache		2K entry
task predictor	two-level adaptive	14-bit global history
	perceptron	8-bit global history, 4096 Pattern History Table

core consists of a number of ALUs, load/store units according to each configuration. For the memory disambiguation, load-store and store-store pairs are inhibited from the speculative

 ISBN: 978-1-61208-466-4

execution when the effective addresses are matched, within or among the cores. The L1 instruction cache and L1 data cache for each core is 64 KB, and it is designed as 2-way set associative. This is because the data cache hit ratio can be degraded by using MESI protocol among multiple cores. For the reference, tasks are predicted using the two-level adaptive prediction scheme. The two-level adaptive task predictor is similar to the two-level adaptive branch prediction scheme where the branch address simply corresponds to the task starting address [11] [12]. In correspondence with the perceptron-based task predictor, the two-level adaptive task predictor employs a 14-bits of global history register and 16,384 items for the pattern history table. For the perceptron-based task predictor, the length of task history register is 8-bits, and the number of pattern history table is set to 4096. The threshold value for the perceptron learning is shown as 2, where *TN* is the length of the task history register. For both predictors, the task address cache has the size of 2048 entries. Since we do not model the main memory, the hit ratio of L2 cache is assumed to be as 100 %.

$$\theta = 2 \times TN + 14 \quad (2)$$

B. The multi-core processor simulator

Figure 5 depicts how the developed simulator works [13]. *Initialize* function initializes all the associated variables, and *Grouping*, *Create_Window*, and *Fetch_One_Instr* function fetches new instructions to fill core tasks every cycle. The

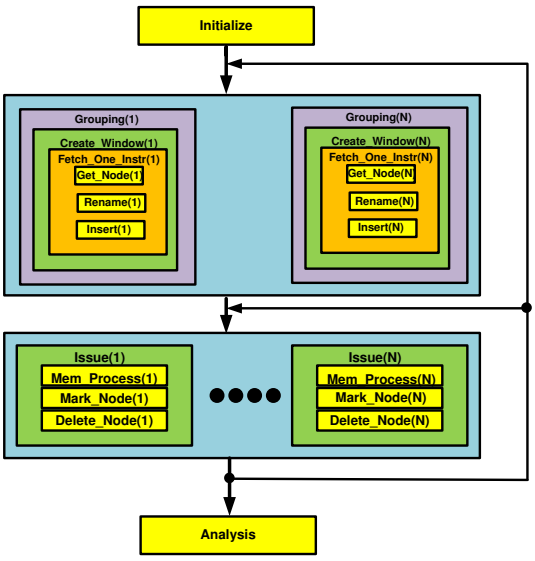


Figure 5. The flow chart of the multi-core processor simulator

instruction fetched by *Get_Node* function is renamed at *Rename* function by receiving timestamps. After the instruction is renamed, it is inserted into a core task by *Insert* function. At the *Issue* function, the instruction in the core task can be retired so long as the corresponding functional unit is available and its time stamp is less than or equal to the current cycle. For implementing the multi-core simulation, *Grouping* function fills instructions of n-core tasks, and *Issue* function deletes instructions according to their timestamps. This process is repeated until all the fetched instructions in the core tasks are deleted to become empty. Then, the core tasks are filled again with instructions by *Grouping* function. Since the cycle is incremented for each process, the core which spends the longest cycles determines the global cycle. If the total number of executed instruction is divided by the number of global cycles spent, then Instruction per Cycle (IPC) can be obtained. The eight SPEC 2000 integer benchmark programs that is used for the input are *bzip2*, *crafty*, *gap*, *gcc*, *gzip*, *mcf*, *parser*, and *twolf* as shown in Table II. The programs are compiled by

TABLE II. SPEC 2000 BENCHMARK PROGRAMS

benchmark	description
bzip2	compression
crafty	chess game
gap	group theory interpreter
gcc	C programming language compiler
gzip	compression
mcf	optimization of combination
parser	word processor
twolf	placement and global routing

SimpleScalar cross C compiler to obtain executables under Linux 3.3.4 [14]. The execution files are again run with SimpleScalar to obtain 100 million MIPS IV instruction traces, which are used as input for the multi-core processors. The task-level parallelism is mapped onto each core, and the trace-driven simulation is performed to get performance [15].

V. The Simulation Results

Figure 6 presents the simulation results of running SPEC 2000 integer programs on the three different task lengths for the single-core, dual-core, quad-core, and octa-core processors. The performance results obtained by the two-level adaptive task predictor and the perceptron-based task predictor are compared in parallel. Figure 6a and 6b are the result of the multi-core processors with the maximum task length of four. Across the number of different cores, $bzip2$ and mcf scores the highest performance owing to the relatively high parallelism and the low cache miss rates. However, gcc results in the lowest performance due to the severe losses from the low instruction and data cache hit rates. For the dual-core processors, the two-level adaptive task predictor brings the geometrical mean of 2.60 IPC, whereas the perceptron-based task predictor results in 2.63 IPC. For the octa-core processors, the two-level adaptive task predictor and the perceptron-based predictor results in 7.64 IPC and 7.73 IPC, respectively. With the perceptron-based task predictor, the performance is 1.7 times enhanced as the number of cores doubles. Therefore, when the performance of the octa-core processor is compared with the single-core, it is 5.4 times higher. With the maximum task length of four, the perceptron-based task predictor performs 1.1 % higher than the two-level adaptive task predictor.

Figure 6c and 6d show the results with the maximum task length of eight. Still, $bzip2$ and gcc show the best and the poorest performance, respectively. For the quad-core processor, the two-level adaptive task predictor brings 7.53 IPC, whereas the perceptron-based task predictor scores 7.75 IPC. The respective performance for the octa-core processor are 12.3 IPC and 12.5 IPC. With the task length of eight, the octa-core processor brings 5.3 times higher performance than the single-core processor, which is slightly lower than the task length of four. However, the task length of eight performs 1.6 times better than the task length of four. Hence, the proposed

scheme scores higher performance than the two-level adaptive scheme by 2.8 %.

Finally, Figure 6e and 6f present the comparison result when the maximum task length is sixteen. $Parser$ outperforms $bzip2$ by the enlargement of the task length, whereas gcc still maintains low performance. For the dual-core processors, the two-level adaptive task predictor brings 6.5 IPC, whereas the perceptron-based task predictor results in 6.9 IPC. For the quad-cores, the respective values are 11.24 IPC and 11.67 IPC. And for the octa-cores, they are increased to 17.8 IPC and 18.3 IPC, respectively. With the perceptron, the performance has increased 1.8 times higher as the single-core goes to the dual-core processor. However, it is slightly decreased to 1.6 times when the quad-cores go to the octa-cores. The octa-core results in the 4.9 times higher performance than the single-core with the maximum task length of sixteen. Although the increase rate has been slowed down, the maximum task length of sixteen gives 1.5 times and 2.4 times higher performances than the maximum task length of eight and four, respectively. When the maximum task length is sixteen, the perceptron-based task predictor prevails the two-level adaptive task predictor by 5.1 %.

VI. Conclusions

In this paper, a perceptron-based task predictor for multi-core processors has been proposed. The single-core to octa-core processors using perceptron with different task lengths have been simulated. As the result shows, the performance of multi-core processors with the perceptron-based task predictor scores higher performance than the two-level adaptive task predictor. When the task lengths are 4, 8, and 16, the respective performance increase over the two-level adaptive scheme are 1.1 %, 2.8 %, and 5.1 %.

For the future research, we will apply the perceptron-based task predictor to the asymmetric multi-core processor to further improve the efficiency, as well as expanding our scope to the multi-core embedded and multi-core digital signal processor architectures.

Acknowledgment

The author would like to thank Hansung University for the financial support of this research.

References

[1] D. E. Culler and J. P. Singh, Parallel Computer Architecture. Morgan Kauffmann Publishers Inc., Aug. 1998.

[2] T. Ungerer, B. Robic, and J. Silk, “Multithreaded Processors,” The Computer Journal, vol. 45, no. 3, 2002.

[3] S. W. Keckler, K. Olukotun, and H. P. Hofsee, Multicore Processors and Systems. Springer, 2009.

[4] M. Monchiero, “How to simulate 1000 cores,” ACM SIGARCH Computer Architecture News archive, vol. 37, no. 2, May 2009, pp. 10–19.

[5] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, “Multi-execution : Multicore caching for data-similar executions,” in Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 164–173.

[6] D. Genbrugge and L. Eckhout, “Chip multiprocessor design space exploration through statistical simulation,” IEEE Transactions on Computers, vol. 58, no. 12, Dec. 2009, pp. 1668–1681.

[7] D. A. Jimenez and C. Lin, “Neural methods for dynamic branch prediction,” ACM Transactions on Computer Systems, vol. 40, no. 2, Mar 1999, pp. 24–36.

[8] ——, “Dynamic branch prediction with perceptrons,” in High Performance Computer Architecture, Jun 2001, pp. 197–206.

[9] D. A. Jimenez, “Fast path-based neural branch prediction,” in Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Dec 2003, pp. 243–252.

[10] T. N. Vijaykumar and G. S. Sohi, “Task selection for a multiscalar processor,” in 31st International Symposium on Microarchitecture, Dec 1998, pp. 81–92.

[11] T.-Y. Yeh and Y. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction,” in Proceedings of the 19th International Symposium on Computer Architecture, May. 1992, pp. 124–134.

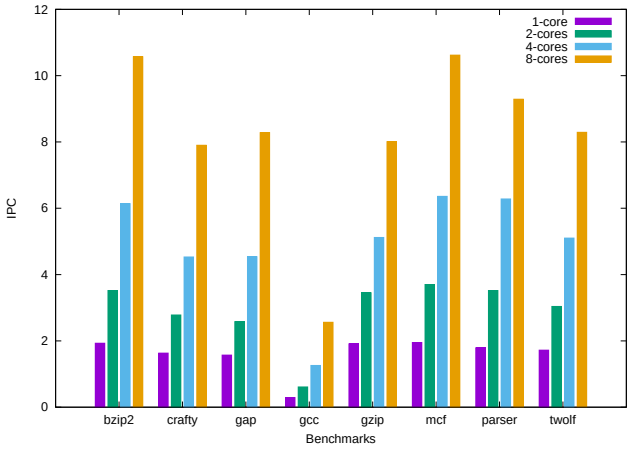
[12] J. Gummaraju and M. Franklin, “Branch prediction in multi-threaded processors,” in Parallel Architectures and Compilation Techniques, Oct 2000, pp. 179–188.

[13] J. Lee, “The study of statistical simulation for multicore processor architectures,” in The Sixth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, Mar 2015, pp. 27–30.

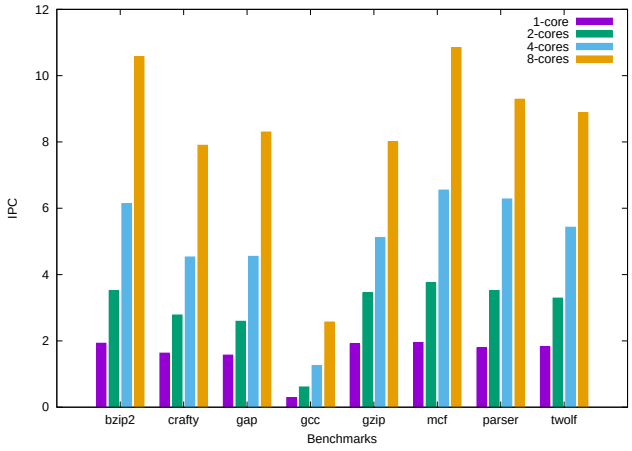
[14] T. Austin, E. Larson, and D. Ernest, “SimpleScalar : An Infrastructure for Computer System Modeling,” Computer, vol. 35, no. 2, Feb. 2002, pp. 59–67.

[15] A.Rico, A. Duran, F. Cabarcas, A. Ramirex, and M. Valero, “Trace-driven simulation of multithreaded applications,” in ISPASS, Apr 2011, pp. 87–96.

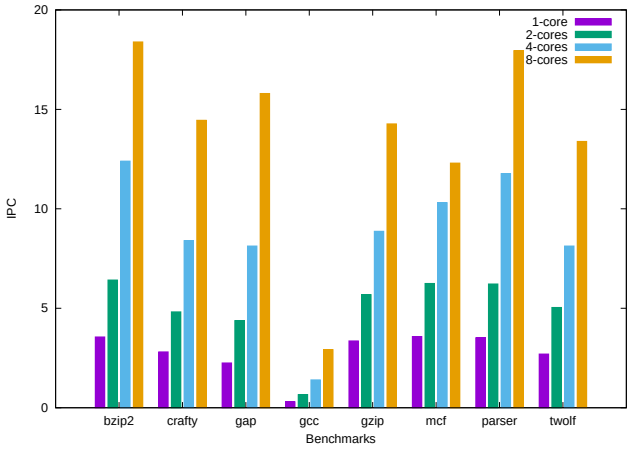
 ISBN: 978-1-61208-466-4



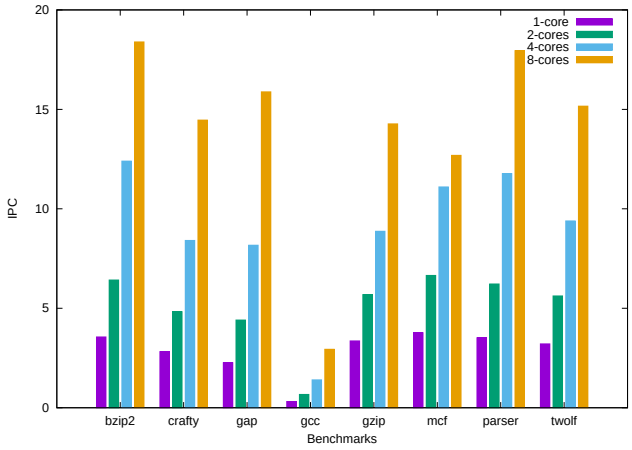
(a) two-level adaptive, maximum task length of 4



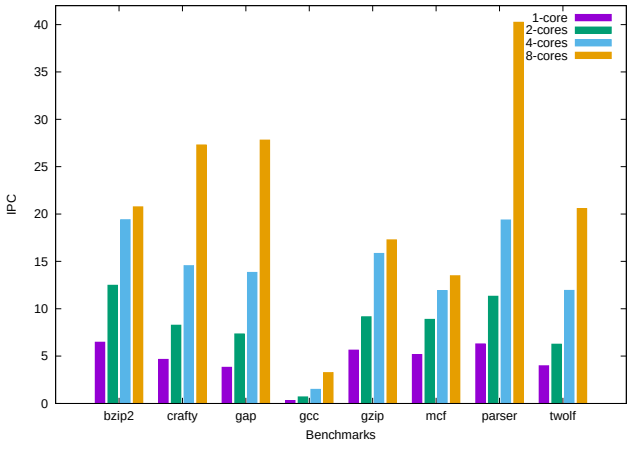
(b) perceptron, maximum task length of 4



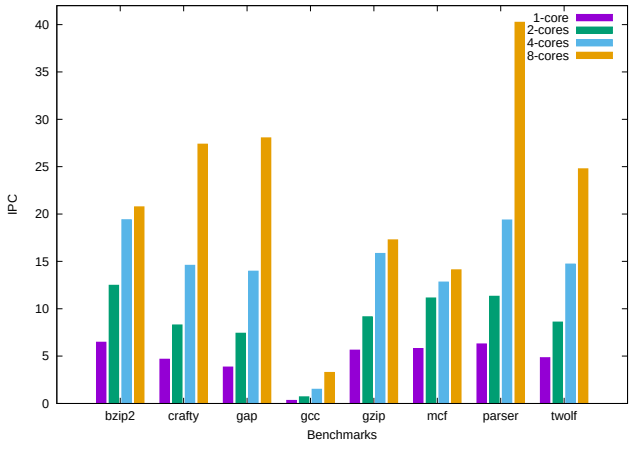
(c) two-level adaptive, maximum task length of 8



(d) perceptron, maximum task length of 8



(e) two-level adaptive, maximum task length of 16



(f) perceptron, maximum task length of 16

Figure 6. Performance results of the two-level adaptive and the perceptron-based task predictor

 ISBN: 978-1-61208-466-4