Postprint: Helper Thread Prefetch Quality Tuning Algorithm Based on Interleaved Prefetch Rate

Authors: Zhang Jianxun, Gu Zhimin

Date: 2018-05-20T00:00:00+00:00

Abstract

Pre-execution helper threads require dynamic prefetch adjustment during the prefetching process, whereas traditional static enumeration methods for controlling parameter values remain unchanged throughout prefetch execution, thereby preventing them from effectively providing quality of service (QoS) guarantees for the main thread. To address this issue, this paper proposes a helper thread prefetch quality parameter adjustment method based on interleaved prefetch rate. First, modeling and analysis of prefetch QoS optimization for helper threads is conducted; second, based on prior interleaved prefetching work, a helper thread parameter value adjustment algorithm based on interleaved prefetch rate is proposed; finally, the proposed helper thread prefetch adjustment algorithm is evaluated and analyzed on a real commercial multi-core platform. Experimental results show that the proposed helper thread prefetch adjustment algorithm achieves a geometric mean performance speedup of 1.114 for benchmark programs, while the traditional static enumeration method achieves a geometric mean performance speedup of 1.135. Experimental results demonstrate that the proposed helper thread prefetch quality adjustment algorithm solves the problem of automatic parameter value adjustment during helper thread prefetching, enabling rapid attainment of prefetch performance improvements comparable to those of static enumeration without requiring static enumeration of parameter values.

Full Text

Preamble

Helper Thread Prefetching Quality Adjustment Algorithm Based on Interleaved Prefetch Rate

Zhang Jianxun[1], Gu Zhimin[2]
(1. College of Information Technology & Engineering, Tianjin University of Technology and Education, Tianjin 300222, China;
2. College of Computer Science, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Pre-execution helper threads require dynamic prefetch adjustment during the prefetching process. However, traditional static enumeration methods keep control parameter values constant throughout prefetch execution, preventing them from effectively providing Quality of Service (QoS) guarantees for the main thread. To address this limitation, we propose a helper thread prefetch quality parameter adjustment method based on interleaved prefetch rate. First, we model and analyze the optimization of helper thread prefetching QoS. Second, building upon prior interleaved prefetching work, we present a helper thread parameter value adjustment algorithm based on prefetch rate. Finally, we evaluate and analyze the proposed helper thread prefetch adjustment algorithm on a real commercial multi-core platform. Experimental results show that the proposed helper thread prefetch adjustment algorithm achieves a geometric mean performance speedup of 1.114 across benchmark programs, compared to 1.135 for the traditional static enumeration method. These results demonstrate that the proposed helper thread prefetch quality adjustment algorithm solves the problem of automatic parameter value adjustment during helper thread prefetching, obtaining performance improvements comparable to static enumeration without requiring manual parameter enumeration.

Keywords: prefetch rate; helper thread; prefetch QoS; dynamic adjustment

0 Introduction

With the development of cloud computing and big data technologies, storage performance in high-performance computing systems has become a critical bottleneck limiting the performance of big data applications. Big data applications require extensive data access operations during algorithm execution, exhibiting irregular and non-uniform memory access patterns that render traditional software and hardware prefetching schemes ineffective. Helper thread prefetching technology has been proposed to hide memory access latency for long-latency instructions. Based on pre-execution concepts, helper thread technology can effectively push data required by the main thread from main memory into the last-level cache of multi-core platforms, transforming program non-sequential locality into algorithmic locality and thereby improving application performance. However, the process of manually enumerating and selecting control parameter values is cumbersome and time-consuming. Therefore, achieving real-time selection and adaptive adjustment of helper thread control parameter values is key to addressing this problem.

Building upon our prior work [2-6], this paper investigates adaptive adjustment

of helper thread prefetching control strategies. To solve the real-time control problem during helper thread prefetch execution, we propose a prefetch-rate-based helper thread parameter adjustment algorithm and conduct systematic evaluation on a multi-core platform. Experimental results demonstrate that the proposed dynamic parameter adjustment algorithm achieves performance improvements similar to manual enumeration without requiring manual parameter enumeration.

1 Related Work

With the widespread adoption of multi-core processors, thread-level pre-execution techniques have attracted significant academic attention [7,8]. Hardware-based approaches require additional hardware modifications to dynamically generate instruction execution fragments in real time. Software-based approaches primarily optimize serial programs at source code and binary levels, requiring offline profiling of program execution semantics. Kim et al. [9] first introduced helper thread pre-execution technology to a real Intel simultaneous multithreading platform, proposing two synchronization mechanisms: loop-based synchronization and sampling-based synchronization to address inter-thread synchronization issues. Song et al. [10] presented a compiler framework for automatic construction of helper thread code. Jung et al. [11] proposed a PV-signal-based helper thread synchronization mechanism. Kamruzzaman et al. [12] introduced an inter-core prefetching scheme where helper threads execute data prefetching on idle cores; after executing for a certain distance, the helper thread and main thread swap CPU cores, with the main thread migrating to the helper thread' s core to benefit from prefetched data while the helper thread continues prefetching on the main thread' s original core. This scheme divides the executing program into equal-length blocks, with block sizes predetermined through offline program profiling. Lu et al. [13] utilized the ADORE dynamic optimization framework to implement helper thread data prefetching. Luo et al. [14] proposed a dynamic performance tuning technique for speculative threads, monitoring hardware performance counters in real time to evaluate execution time of the original serial program during speculative thread execution and making decisions about speculative thread effectiveness based on evaluation results. Fei et al. [15] proposed a gradient optimization method for helper thread control parameters based on interleaved prefetching. Andrew et al. [16,17] proposed a PID (proportion integration differentiation) control-based application QoS technique for shared resource usage in multi-core platforms. This technology employs a PID control mechanism that monitors application shared resource usage through hardware performance counters, then restricts resource usage through methods like frequency reduction and clock modulation to provide QoS guarantees for different application types running on multi-core platforms. The work of Luo and Andrew provides valuable techniques and ideas for implementing dynamic QoS adjustment for helper thread prefetching.

In summary, traditional helper thread research determines control parameter values through offline profiling and empirical selection. This manual parameter selection process relies heavily on experience and involves substantial effort, making it impractical for real applications. This paper investigates online parameter value selection and adjustment for interleaved prefetching helper thread control parameters K, P, and B, implementing a real-time adaptive parameter selection adjustment algorithm.

2 Helper Thread Prefetching QoS Optimization Model

Helper threads introduce three control parameters—prefetch distance K, prefetch size P, and synchronization block count B—to regulate and control helper thread prefetching behavior. For descriptive convenience, we provide a formal description of the helper thread prefetching QoS strategy optimization problem.

Assume Q is a set containing helper thread prefetching QoS strategies, Q = {Q_i}, i = 1, 2, 3, ⋯, m, where Q_i = (K, P, B) is a triple with K, P, and B representing the three control parameter values for interleaved prefetching helper threads. We refer to Q_i as a helper thread prefetching QoS strategy. Here, K represents the prefetch distance by which the helper thread leads the main thread, measured in number of hot loops; P represents the workload at which the helper thread begins prefetching for the main thread, also measured in loop count; and B represents how frequently the helper thread synchronizes with the main thread, similarly measured in loop count. The primary purpose of synchronization is to prevent the helper thread from deviating from the main thread' s execution path—either lagging behind or running too far ahead.

Definition 1: Interleaved Prefetch Rate R_p. The prefetch rate in helper threads is defined as R_p = P/(K+P), where R_p (0,1). Interleaved prefetching refers to the technique where, when the main thread enters a hot loop, the helper thread does not prefetch all hot data to avoid lagging behind the main thread. Instead, it prefetches data for the main thread in a skipping manner, leaving part of the hot data access to the main thread, thereby enabling the main thread and helper thread to share long-latency memory access instructions. Their memory access data streams interleave and execute in parallel, hence the term interleaved prefetching.

As evident from the prefetch rate definition, when the prefetch distance is known, the prefetch workload size can be adjusted through the prefetch rate R_p.

Definition 2: Prefetch QoS Evaluation Function. The evaluation function is defined as Eval(Q_i, S_i) = (1-) × IPC_i, where [0,1]. The evaluation function takes two input parameters: a prefetch QoS strategy Q_i and a hot module execution sample S_i. IPC_i represents the main thread performance metric obtained after applying prefetch QoS strategy Q_i during the i-th sampling period.

Assume the main thread hot module Γ is an instruction stream, Γ = {i_1, i_2,

i_3, ···, i_N}. Let U be the hot module execution sample size, measured in loop count. The program hot module' s instruction stream Γ is divided into instruction sequences of length U, i.e., Γ = {S_i}, i = 0, 1, 2, ···, L, where |S_i| = U and S_i is a hot module sampling execution sample.

Helper thread prefetching QoS optimization can be expressed as: max_{1 i m} Eval(Q_i, S_i), meaning that within the hot module' s instruction stream, by selecting instruction sequences of length U as hot module sampling execution samples, applying corresponding prefetch QoS strategy Q_i to each hot module execution sample S_i during program execution, and calculating CPI_i for each sample through the evaluation function after sampling, the interleaved prefetching QoS strategy with the minimum CPI value is ultimately selected.

From Definition 2, Eval(Q_i, S_i) = (1-) × IPC_i = _i, which evaluates the performance of hot module execution sample S_i. A larger _i value indicates lower clock cycles per instruction (CPI) for the main thread' s processor and better main thread performance.

Definition 3: Let _i be the performance evaluation result of sampling sample S_i by the prefetch adjustment framework, and _j be the performance evaluation result of sampling sample S_j. Then _{i,j} = _i - _j is called the main thread performance change from prefetch QoS strategy Q_i to Q_j, where i = 1, 2, ···, j = 1, 2, ···. Here, _{i,j} represents the difference in performance improvement between two different prefetch QoS strategies on two hot module sampling samples. If _{i,j} > 0, prefetch QoS strategy Q_j is preferred over Q_i; if _{i,j} 0, the two prefetch QoS strategies provide equivalent performance improvement; if _{i,j} < 0, prefetch QoS strategy Q_i is preferred over Q_j.

Definition 4: After hot module sampling execution samples from S_0 to S_c adopt prefetch QoS strategies Q_0 to Q_c respectively, if _{0,1} > 0, _{1,2} > 0, ···, _{c-1,c} > 0, the main thread performance improvement shows an improving trend. Conversely, if _{0,1} < 0, _{1,2} < 0, ···, _{c-1,c} < 0, the main thread performance improvement shows a deteriorating trend. Here, c is referred to as the performance gradient trend threshold.

3 Helper Thread Adjustment Algorithm Based on Prefetch Rate

The essence of helper thread prefetch adjustment is to control helper thread prefetching behavior by adjusting the helper thread' s prefetch workload. Prefetch distance K and prefetch workload P are two correlated variables whose relationship can be described by prefetch rate R_p. With a given prefetch distance, an appropriate R_p value can always be found to achieve relatively optimal main thread performance. Traditional enumeration methods that adjust individual variables separately result in slow algorithm convergence and cannot quickly find relatively locally optimal parameter value combinations. This section introduces a helper thread prefetch adjustment algorithm based

on prefetch rate R_p that rapidly traverses and generates parameter value combinations within a specified range according to R_p, then applies each generated parameter value combination as a prefetch QoS strategy to the helper thread for performance evaluation during the parameter training period.

3.1 Determining Parameter Value Upper Bounds

The outer loop count of hot loops in the hot module limits the maximum values of K, P, and B parameters. Assuming the outer loop count of hot loops is L_max, the parameter values satisfy the following constraints: a) 0 K L_max, 0 P L_max - K, 0 B × (K + P) L_max b) 0 B (L_max / (K + P))

For interleaved prefetching helper threads, Huang et al. [18,19] proposed a cache-correlation-based prefetch distance evaluation strategy primarily used to assess the value range of prefetch distance K. Cache correlation is a concept that maps hot loops to the set associativity of the last-level shared cache in multi-core platforms to evaluate prefetch distance. The core idea is that for a hot loop, if after executing the j-th iteration the data accessed by the hot loop has filled the cache, continuing to access hot loop data will trigger cache replacement. In this case, the cache correlation of the hot loop relative to the shared cache is j. The core idea of interleaved prefetching helper threads is that the main thread and helper thread jointly share the LLC cache miss instructions.

Assume the helper thread can lead the main thread by at most K_max loops. When the main thread completes hot data access for the first K_max loops, since the helper thread has less code than the main thread, the helper thread has prefetched at least K_max loops of hot data. If 2 × K_max exceeds the cache correlation j, subsequent prefetching of hot data by the helper thread may cause cache pollution. Therefore, K should be less than or equal to j/2 (taking K_max = j/2). This theoretical assumption is that the helper thread and main thread share LLC cache miss instructions, so the prefetch size P at this moment is j/2 after running K_max hot data.

Cache correlation can be obtained through profiling experiments. Based on the cache correlation concept, prefetch distance K and P can also be predicted statically. Assuming each hot loop in the hot module accesses Loopsize bytes of cache miss data and the LLC shared cache capacity is Cachesize, then K_max (Cachesize / Loopsize) / 2. The static prediction method primarily divides the shared cache capacity into two parts shared by the helper thread and main thread. If the number of hot loops L in the main thread exceeds (Cachesize / Loopsize) / 2, the cache is already full, and further prefetching by the helper thread will cause cache pollution. Therefore, the upper bounds for K_max and P_max parameters are set to (Cachesize / Loopsize) / 2.

The synchronization parameter B represents the synchronization distance—how many block intervals between synchronizations—and is generally set to 1, meaning synchronization after each block. The maximum upper bound for parameter B is (L_max / (K + P)).

3.2 Helper Thread Prefetch Adjustment Algorithm Based on Prefetch Rate

Helper thread prefetch rate R_p adjusts helper thread prefetching behavior. With a given K value, traversing prefetch rate R_p significantly reduces parameter optimization time. The value range of R_p is (0, 1), and prefetch size P can be calculated through the formula P = R_p × K / (1 - R_p). The parameter value generator algorithm is shown in Algorithm 3-1.

Algorithm 3-1: Helper Thread Prefetch Adjustment Algorithm Based on Prefetch Rate

```
Input: (K', Rp', CLAR, SYNC)
Output: (K, P)
01: init parameter K = K', Rp = Rp';
02: If (SYNC == 1) then      // synchronization block count is 1
03:     if (CLAR > 0) then    // CLAR > 0 indicates sufficient main thread computation workl
04:         K = K + Step;      // adjust K value
05:         P = Rp * K / (1 - Rp);
06:     else
07:         P = P + Step;
08:     Endif
09: Else
10:     // synchronization block not 1, need to adjust synchronization parameters
11:     // based on current K and P
12: Endif
```

Algorithm 3-1 generates parameter values under several conditions: a) By default, parameter B = 1, meaning synchronization after each block. b) CYLR indicates the main thread' s computation workload size. If CLAR > 0, the main thread has sufficient workload, and K is set to 0 during parameter generation while adjusting P. c) If the CLAR flag is 0, indicating insufficient main thread workload, both parameters K and P need adjustment. K and P adjustment is based on traversing the prefetch rate. The main process involves first specifying a K value, then calculating the corresponding P value as R_p varies, returning the result. Afterward, the prefetch rate resets to 0, K increases by step size STEP to K' , and the corresponding P value is calculated and returned based on the R_p value. d) The SYNC flag indicates whether to adjust the synchronization parameter B. Generally, the synchronization block count is set to 1 (B = 1). If the SYNC flag is not 1, it means the synchronization distance needs to be increased based on current K and P values, adjusting only parameter B.

Let Policy_num represent the number of helper thread prefetch QoS strategies and U represent the size of hot module execution samples. Each sampling sample is used to evaluate one prefetch QoS strategy. Therefore, the total sample size for the helper thread prefetch QoS strategy training period is Policy_num × U. Larger total training samples leave less optimization space for helper thread prefetching, so the prefetch parameter adjustment algorithm must

converge quickly, requiring limits on the number of generated parameter values.

4 Experimental Evaluation

We validate and evaluate the helper thread prefetch adjustment algorithm on a real commercial machine. The experimental platform uses an Intel Core 2 Quad Q6600 processor with detailed system configuration shown in Table 1 . The main thread and helper thread are bound to fixed CPU cores through the Linux Affinity interface. The hardware prefetchers on Q6600' s four CPU cores remain enabled.

Table 1 : Q6600 Experimental Platform Configuration - Processor: Intel Core 2 Quad Processor Q6600 - Memory: 2 GB (DDR 667, non-ECC) - L1 D-Cache: 32 KB × 4, 8-way set-associative, cache line 64 bytes - L1 I-Cache: 32 KB × 4, 8-way set-associative, cache line 64 bytes - L2 Cache: 4096 KB × 2, 16-way set-associative, cache line 64 bytes - FSB Speed: 1066 MHz - Compiler: Gcc version 4.3.0 -O2 - OS: Fedora 9 with kernel 2.6.34

The benchmark suite consists of seven scientific computing programs: Mst, Em3d, Tsp, Health, Mcf from SPEC CPU 2006, Gcc, and Libquantum. Table 2 presents the input sets and hot modules for test programs. The last column of Table 2 shows the best manually-set parameter values for prefetch helper threads. Except for 426.Libquantum, all benchmark input sets use reference input sets. We compare dynamic parameter adjustment with static prefetching and the traditional PV [9] helper thread method.

Table 2: Benchmark Programs and Prefetch Control Parameter Values | Benchmark | Input Set | Hot Module | Manual Interleaved Prefetch Parameters (K-P-B) | |———-|———-|———|———————————-| | 429.Mcf | SPEC2006 Ref/inp.in | refresh_potential | 24-24-nosyn | | 403.Gcc | SPEC2006 Ref/166.i | Primal_bea_mpp | 5-290-nosyn | | Libquantum | quantum_cnot | Reg_is_remote_constant_p | 192-50-nosyn | | | quantum_sigma_x | | | | | quantum_toffoli | | | | Olden | HashLookup | | 100-30-nosyn | | Olden | Fill_from_fields | | | | Olden | 10,000,000 | | | | Merge | Health | | | | Olden | 5000 iters | | | | | 5 levels, Check_patients_waiting | | |

Table 3 : Hot Module Invocation Information and Initial Parameter Values | Program Hot Module | Invocations | Initial K | K_max | |————————-|———-|———-|———-| | fill_from_fields | | | | | HashLookup | | | | | refresh_potential | | | | |

4.1 Experimental Platform and Benchmark Programs

To eliminate system interference during program execution, all test programs were run seven times, with the median execution time taken as the final evaluation result. Figure 1 [Figure 1: see original paper] shows the normalized execution time for all test programs relative to the original serial program baseline, with HPCF representing the proposed prefetch adjustment framework.

Compared with the traditional PV helper thread method, manual interleaved prefetching achieves the best results with an average performance speedup of 1.135 across benchmarks. The automatic parameter selection interleaved prefetching method achieves a geometric mean performance speedup of 1.114, while the traditional PV method achieves 1.020. The automatic parameter selection method produces performance close to manual selection, with acceptable performance differences. The main difference between real-time parameter adjustment and manual parameter setting lies in the overhead of parameter training periods and real-time sampling. However, as problem scale increases, the gap between real-time parameter selection and manual methods becomes minimal. For example, Mcf' s serial program runtime exceeds 400 seconds; the dynamic parameter adjustment method achieves a main thread speedup of 1.226, while manual parameter setting achieves 1.232.

4.2.1 Evaluation of Helper Thread Prefetch Adjustment

Since Mcf, Mst, and Em3d show significant performance improvements with interleaved prefetching, this section uses these three benchmarks to analyze the impact of K_step on the prefetch adjustment framework performance. Benchmark programs were run seven times, with the median taken as the evaluation result to eliminate system fluctuations. Table 3 shows hot module invocation counts and initial K values and upper bounds for the three benchmarks. The parameter value generation algorithm execution process is: dividing the prefetch distance K range [0, K_max] into equal segments with step size K_step, selecting one K value per segment, then generating parameter P based on prefetch rate R_p variations. In the initial algorithm implementation, R_p ranges from 0.1 to 0.9 with step size 0.1, generating nine (K, P) parameter combinations for each fixed prefetch distance K value. This discussion focuses on the impact of K_step on prefetch adjustment framework performance. With given initial K value K_init, upper bound K_max, and step size K_step, the number of parameter combinations generated by the parameter value generator is P_num = ((K_max - K_init) / K_step) × 9.

Figures 2-4 [Figure 2: see original paper][Figure 3: see original paper][Figure 4: see original paper] show the performance speedups of Mst, Mcf, and Em3d programs with the prefetch adjustment framework without feedback mechanisms under different K_step values.

The figures demonstrate that as K_step increases, the number of parameter value combinations generated for controlling helper threads gradually decreases. When K_step increases beyond a certain value, the parameter value generator produces only one K value because further increasing the step size exceeds K' s upper bound K_max. For example, in Figure 2 [Figure 2: see original paper], when Mst' s step size exceeds 1600, the prefetch adjustment framework uses only (K_init + K_step) as the K value, generating nine parameter value combinations through R_p variation as candidate values. Similarly, in Figure 3 [Figure 3: see original paper], when Mcf' s K_step exceeds 750, the parameter value

generator produces only one K value, resulting in nine parameter value combinations from R_p variation. Therefore, K_step cannot be too large, as excessive step size reduces the candidate parameter value set and loses opportunities to find locally optimal parameter values.

Except for Em3d, the other two programs show a decreasing performance speedup trend as K_step increases, because the parameter value generator produces fewer combinations that are not locally optimal. However, when K_step is too small, excessive parameter value generation prolongs the parameter training period and degrades program performance. For example, in Figure 2, when Mst' s K step size is 40, the K_max upper bound of 3150 produces 78 K value partitions, with the parameter value generator creating 702 parameter value combinations. Assuming the parameter selection phase uses one hot module invocation as a sampling sample, 702 hot module invocations are needed to complete the first-stage parameter training to select 78 parameter value combinations for the second-stage evaluation. If the sampling sample size increases to 10 hot module invocations during the reselection phase, another 780 sampling training iterations are required. When K_step is 40, the total parameter training requires 702 + 780 = 1410 hot module invocations. As Table 3 shows, Mst' s hot module is invoked 10,000 times total, making the parameter training period 14.1% of total hot execution time, which impacts performance improvement due to the lengthy training period with small step sizes.

5 Conclusion

Helper thread prefetching strategies introduce three control parameters—prefetch distance K, prefetch size P, and synchronization size B—to regulate and control helper thread prefetching behavior. The search space for K-P-B parameter values is typically enormous, making exhaustive search through empirical manual selection impractical. This paper proposes a dynamic adjustment algorithm for helper thread prefetching based on interleaved prefetch rate, enabling real-time online adaptive adjustment of helper thread prefetching control parameter values. Experiments on an Intel Core 2 Quad Processor Q6600 with scientific computing benchmarks Em3d, Mst, and Mcf from SPEC CPU benchmark 2006 demonstrate that the proposed helper thread adaptive prefetching framework can quickly obtain performance comparable to manual enumeration without requiring manual parameter enumeration.

References

[1] Luk C K. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors [C]// Proc of the 28th Annual International Symposium on Computer Architecture. New York: ACM Press, 2001: 40-51.

[2] Gu Z, Zheng N, Zhang Y, et al. The stable conditions of a task-pair with helper-thread in CMP [C]// Proc of International Conference on Parallel and Distributed Processing Techniques and Applications. 2009: 125-130.

[3] Gu Z, Fu Y, Zheng N, et al. Improving performance of the irregular data intensive application with small workload for CMPs [C]// Proc of the 40th International Conference on Parallel Processing Workshops. 2011: 279-288.

[4] Huang Y, Tang J, Gu Z, et al. The performance optimization of threaded prefetching for linked data structures [J]. International Journal of Parallel Programming, 2012, 40(2): 141-163.

[5] Zhang Jianxun, Gu Zhimin. Real-time online evaluation method for helper thread prefetching quality [J]. Computer Applications, 2017, 37(1): 114-119.

[6] Zhang Jianxun, Gu Zhimin, Hu Xiaohan, et al. Multi-core helper thread prefetching method for irregular big data analysis applications [J]. Journal on Communications, 2014, 35(8): 137-146.

[7] Byna S, Chen Y, Sun X. A Taxonomy of data prefetching mechanisms, TR IIT//CS-SCS07-01 [R]. 2007.

[8] Ou Guodong. Research on thread-based data prefetching technology [D]. Changsha: National University of Defense Technology, 2007.

[9] Kim D, Steve S L, Wang P H, et al. Physical experimentation with prefetching helper threads on intel' s hyper-threaded processors [C]// Proc of International Symposium on Code Generation and Optimization. 2004: 27-38.

[10] Song Y, Kalogeropulos S, Tirumalai P. Design and implementation of a compiler framework for helper threading on multi-core processors [C]// Proc of the 14th International Conference on Parallel Architectures and Compilation Techniques. 2005: 99-109.

[11] Lee Jaejin, Jung Changhee, Lim Daeseob, et al. Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems [J]. IEEE Trans on Parallel and Distributed Systems, 2009, 20(9): 1309-1324.

[12] Kamruzzaman M, Swanson S, Tullsen D M. Inter-core prefetching for multicore processors using migrating helper threads [C]// Proc of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems. 2011: 393-404.

[13] Lu Jiwei, Das A, Hsu W C, et al. Dynamic Helper threaded prefetching on the Sun UltraSparc CMP processor [C]// Proc of the 38th Annual IEEE//ACM International Symposium Microarchitecture. New York: ACM Press, 2005: 93-104.

[14] Luo Yangchun, Packirisamy V, Hsu Weichung, et al. Dynamic performance tuning for speculative threads [C]// Proc of the 36th International Symposium on Computer Architecture. New York: ACM Press, 2009: 462-473.

[15] Pei Songwen, Zhang Junge, Ning Jing. Gradient learning parameter control helper thread prefetching model [J]. Journal of National University of Defense Technology, 2016, 38(5): 59-63.

[16] lllikkal R, Chadha V, Herdrich A, et al. PI-RATE: QoS and Performance Management in CMP Architectures [J]. Newsletter ACM SIGMETRICS Performance Evaluation Review, 2010, 37(4): 3-10.

[17] Andrew H, Ramesh I, Ravi I, et al. Rate-based qos techniques for cache//memory in CMP platforms [C]// Proc of the 23th Annual International Conference on Supercomputing. 2009: 479-488.

[18] Huang Y, Gu Z, Tang J, et al. Estimating effective prefetch distance in threaded prefetching for linked data structures, international journal of parallel programming [J]. 2012, 40(5): 465-487.

[19] Huang Yan, Zhang Qikun, Duan Zhaolei, et al. Thread data prefetch distance control strategy based on cache behavior characteristics [J]. Journal of Electronics & Information Technology, 2015, 37(7): 1633-1638.

Note: Figure translations are in progress. See original paper for figures.