行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※
※　　　　　　　　　　　　　　　　　　　　　　　※
※　　預先擷取鏈結資料結構之可程式化記憶體階層　　※
※　　　　　　　　　　　　　　　　　　　　　　　※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：x 個別型計畫　　□整合型計畫

計畫編號：NSC　91-2213-E-002- 056 -

執行期間：　　91 年　1　月　1　日至　91 年　7　月 31　　日

計畫主持人：楊佳玲

共同主持人：

計畫參與人員：曾宏偉 楊善詠 蘇夢昌 費懷忠

本成果報告包括以下應繳交之附件：

□赴國外出差或研習心得報告一份

□赴大陸地區出差或研習心得報告一份

□出席國際學術會議心得報告及發表之論文各一份

□國際合作研究計畫國外研究報告書一份

執行單位：台灣大學資訊工程學系

中　華　民　國　91　年 10　月 28 日

行政院國家科學委員會專題研究計畫成果報告

預先擷取鏈結資料結構之可程式化記憶體階層

A Programmable Memory Hierarchy for Prefetching Linked Structures

計畫編號：NSC 91-2213-E-002-056-
執行期限：91 年 1 月 1 日至 91 年 7 月 31 日

主持人：楊佳玲　　台灣大學資訊工程系
e-mail: yangc@csie.ntu.edu.tw
http://www.csie.ntu.edu.tw/~yangc

一、中文摘要

由於中央處理器與記憶體間效能的差距日益擴大，如何減輕此差距對高性能電腦系統建構上的影響，成為一個重要的課題。快取記憶體的應用被公認為可以非常有效的改進記憶體系統效能。然而，其有效性往往會因為程式本身對於快取記憶體取用能力不佳而受到侷限。因此，發展隱藏記憶體存取時間的技術，對於彌補處理器與記憶體間效能差距是非常重要的。

預先擷取常被運用來重疊運算與資料的存取。對於以矩陣為主的應用程式，預先擷取技術在過去十年間已經發展相當成熟。但是如何運用預先擷取技術在鏈結資料結構為主的應用程式卻仍然是一個極具挑戰性的問題。其原因在於鏈結資料結構在記憶體位址的配置上並不如矩陣那樣的規律，以及指標追逐為問題。

在這個計畫中，我們針對以鏈結資料結構為主的程式，建立一個上推模式預先存取架構。在此架構中，我們在記憶體階層中的每一層裡安裝一個可程式化的預先擷取引擎。 這些預先擷取引擎協調地執行存取鏈結資料結構的指令。被這些預先擷取引擎存取的資料會主動推進至記憶體階層中的頂層。在計畫執行期間，我們發展出此架構之模擬器並對一些以鏈結資料結構為主的應用程式加以測試。實驗結果顯示此預先擷取技術最高可消除 100%的記憶體延滯時間，總執行時間平均減少 19%。

關鍵詞：預先擷取、鏈結資料結構、快取記憶體、記憶體階層

Abstract

The widening performance gap between processors and memory makes techniques that alleviate this disparity essential for building high-performance computer systems. Caches are recognized as a cost-effective method to improve memory system performance. However, a cache's effectiveness can be limited if programs have poor locality. Thus techniques that hide memory latency are essential to bridging the CPU-memory gap.

Prefetching is a commonly used technique to overlap memory accesses with computation.Prefetching for array-based numeric applications with regular access patterns has been well studied in the past decade. However, prefetching for pointer-intensive applications remains a challenging problem. Prefetching linked data structures (LDS) is difficult because address sequences do not present the same arithmetic regularity as array-based applications and because data dependence of pointer dereferences can serialize the address generation process (the pointer-chasing problem).

In this project, we built a generic prefetching framework for pointer-based applications based on the previously proposed push model. We use a programmable processor, a prefetch engine (PFE), at each level of the memory hierarchy to cooperatively execute instructions that traverse a linked data structure. Cache blocks accessed by the processors at the L2 and memory levels are pro-actively pushed up to the CPU. Simulation results show that the proposed prefetching scheme can reduce up to 100% of the memory stall time on a suit of pointer-based applications, reducing overall execution time by an average 19%.

Keywords: prefetching, pointer-based applications, cache, the memory hierarchy

二、Introduction & Objective

Microprocessor performance has been growing at a rate of 60% per year in the past decade. However, memory access time is increasing at a much slower rate, at about 7% per year. Current DRAM (Dynamic Ram) implementations usually take a few hundred naroseconds to retrieve data. As the performance gap between processors and memory continues to grow, techniques that reduce the effect of this disparity are essential to building a high-performance computer system.

The use of caches between the CPU and main memory is recognized as an effective method to bridge this gap. The design of caches is based on one important program property - locality of references. If programs exhibit good locality, the majority of memory requests can be satisfied by caches without having to access main memory. However, a cache's effectiveness can be limited for programs with poor locality. For applications with regular access patterns several compiler techniques (e.g. blocking and loop transforms)1 can be used to improve program locality. However, it is difficult for compilers to perform such optimization for applications with irregular access patterns. Thus techniques that hide memory latency are important in addition to the use of caches.

The idea of hiding memory latency is allowing CPU execution and memory accesses to proceed in parallel. One commonly used latency hiding technique is prefetching. Prefetching for array-based numeric applications with regular access patterns has been well studied [1][2][3][4]. However, prefetching for pointer-intensive applications remains a challenging problem.

Conventional prefetching schemes rely on address stream regularity to predict future addresses. Many scientific computing applications, which use array data structures, present such regularity. Unfortunately, commercial applications, such as databases, graphics and VLSI applications, usually create sophisticated data structures using pointers and do not exhibit sufficient regularity for conventional prefetch techniques to exploit. Furthermore, prefetching pointer-intensive data structures can be limited by the serial nature of pointer dereferences---called the pointer-chasing problem---since the address of the next node is not known until the contents of the current node are accessible. The pointer-chasing problem makes it difficult to schedule prefetch requests far enough ahead to actually hide memory latency.

In this project, we develop a programmable memory hierarchy that is able to perform prefetching for pointer-based applications based on the previously proposed data movement model – push [6]. The push model performs pointer dereferences at lower levels of the memory hierarchy and pushes data up to the processor. This decouples the pointer dereference from the transfer of the current LDS element up to the processor. Implementations can pipeline these two operations and eliminate the request-response delay required for a conventional pull-based technique where the processor fetches an LDS element before requesting the next element. To realize this push model, a prefetch engine (PFE) is attached to each level of the memory hierarchy. The prefetch engine executes instructions that access LDS elements, and cache blocks accessed by the prefetch engines are pushed up to the CPU.

The existing push-based prefetching framework performs a preliminary performance evaluation using a limited implementation of the push model. The initial design can only support linked-list traversals, which simplifies the interaction among prefetch engines. In this project, we build a flexible implementation of the push model, which is capable of performing prefetching for a multitude of LDS traversal kernels.

Simulation results show that the proposed prefetching scheme can reduce up to 100% of the memory stall time on a suit of pointer-based applications, reducing overall execution time by an average 19%. We have published the results in the 4th International Symposium on High Performance Computing (ISHPC-IV) [7].

三、Results Summary

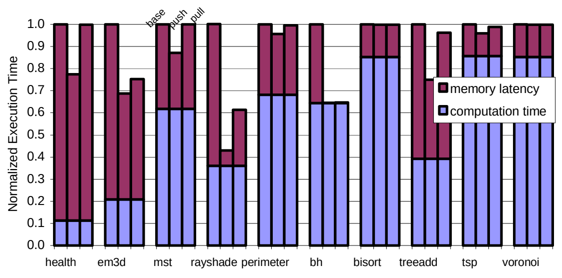

Figure 1: Performance comparison between the push and pull model

In this section, we summary the simulation results based on the Olden Benchmark [8]. Figure1 shows execution time normalized to the base system without prefetching. For each benchmark, the three bars correspond to the base, push and pull models, respectively. Execution time is divided into 2 components, memory stall time and computation time. We obtain the computation time by assuming a perfect memory system. For the set of benchmarks with tight traversal loops (health, mst, treeadd), the push model is able to reduce between 25% and 38% of memory stall time (13% to 25% overall execution time reduction) while the pull model can only reduce the stall time by at most 4%. Perimeter traverses down a quad-tree in depth-first order, but has an unpredictable access pattern once it reaches a leaf. Therefore, we only prefetch the main traversal kernel. Although perimeter performs some computation down the leaves, it has very little computation to overlap with the memory access when traversing the internal nodes. So the pull model is not able to achieve any speedup, but the push model reduces the execution time by 4%.

For applications that have longer computation lengths between node accesses (bh, rayshade, em3d), we expect larger reductions in memory stall time than for programs with little computation between node accesses. From Figure 1 we see that the push model performs close to a perfect memory system for rayshade and bh (89% and 100% memory stall time reduction), reducing execution time by 57% and 36%, respectively. The pull model achieves similar improvements for bh, but reduces execution time by only 39% for rayshade. The exception is em3d, which shows only a 31% reduction in memory stall time and 25% in execution time, with similar performance for the pull model. That is because Em3d has poor L1 cache performance (57% load miss rate), but the L2 cache is able to capture 80% of these misses. Bisort and tsp dynamically change the data structure while traversing it so the prediction accuracy is low for this type of application is low. For tsp, we are able to identify some traversal kernels that do not change the structure dynamically. The results presented here prefetch only these traversal kernels. The push model is able to reduce the execution time by 4% and the pull model 1%. For bisort, neither the push or pull model is able to improve performance because the prediction accuracy is low (only 20% of prefetched cache blocks are useful). By only prefetching one node ahead, both the push and pull can reduce the execution time by 3%. Voronoi uses pointers, but array and scalar loads cause most of the cache misses. So we are not able to see any performance improvement for either the push or pull model.

四、Conclusion

In the project, we have successfully built a programmable memory hierarchy for prefetching linked-data structures. We showed the effectiveness of the proposed scheme through simulations. We have published this work in the 4th International Symposium on High Performance Computing (ISHPC-IV)[7].

五、 Acknowledge

Students who join this projects are Houng-Wei Tseng, Hsun-Young Yang, Men-Chan Shu and Hwi-Cheng Fei.

六、Bibliography

1. M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. *In Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pages 30-44, June 1991.
2. Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 363-363, May 1990.
3. Tien-Fu Chen and Jean-Loup Baer. Reducing Memory Latency via Non-Blocking and Prefetching Caches. *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51-61, 1992.
4. A. K. Porterfield. Software Methods for Improvement of Cache Performance on Supercomputer Applications. Ph.D Thesis, Department of Computer Science, Rice University, May 1989.
5. T.C. Mowry and M. S. Lam and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating System*, pages 62-73, October 1992.
6. C. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked-Data Structures. *In Proceedings of the ACM International Conference on Supercomputing*, pages 176-186, May 2000
7. C. Yang and A. R. Lebeck A Programmable Memory Hierarchy for Prefetching Linked Data Structures, *in Proceedings of the 4th*

International Symposium on High Performance Computing (ISHPC-IV), Springer-Verlag, May 2002, Japan

8．A. Roger, M. Carlisle, J. Reppy and L. Hendren. Supporting Dynamic Data Structures in Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems 19 (1995)*