Information Sciences 192 (2012) 39–49

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A refactoring method for cache-efficient swarm intelligence algorithms

Feng-Cheng Chang [a], Hsiang-Cheh Huang [b,*]

[a] Dept. of Innovative Information and Technology, Tamkang University, Taiwan
[b] Dept. of Electrical Engineering, National University of Kaohsiung, Taiwan

ARTICLE INFO

Article history:
Received 17 August 2009
Received in revised form 24 February 2010
Accepted 25 February 2010
Available online 4 March 2010

Keywords:
Cache
Memory hierarchy
Miss rate
Swarm intelligence
Particle swarm optimization
Genetic algorithm

ABSTRACT

With advances in hardware technology, conventional approaches to software development are not effective for developing efficient algorithms for run-time environments. The problem comes from the overly simplified hardware abstraction model in the software development procedure. The mismatch between the hypothetical hardware model and real hardware design should be compensated for in designing an efficient algorithm. In this paper, we focus on two schemes: one is the memory hierarchy, and the other is the algorithm design. Both the cache properties and the cache-aware development are investigated. We then propose a few simple guidelines for revising a developed algorithm in order to increase the utilization of the cache. To verify the effectiveness of the guidelines proposed, optimization techniques, including particle swarm optimization (PSO) and the genetic algorithm (GA), are employed. Simulation results demonstrate that the guidelines are potentially helpful for revising various algorithms.

1. Introduction

Personal computers (PCs) have become more powerful. On the one hand, high-end PCs are capable of offering a moderate amount of computational capability in academic research and industrial applications. On the other hand, a home-use multimedia PC can provide high quality audio-visual effects for entertainment purposes. When we examine the detailed design of modern computers, we see that there are a number of enhancements to the original von Neumann machine [3]. For instance, the instruction pipeline overlaps the processing and execution within several instructions, and it thus enhances the utilization of the hardware components [6]; the memory hierarchy adopts various techniques to shorten the average memory-access time [1,11]; and the branch detection circuit tries to "guess" the right execution path to reduce the pipeline-restart and the cache-flush [8].

In addition to the advances in computer hardware design, the networking among computers has also evolved. The demand for higher computational capability is increasing steadily. A typical approach is to decrease the response time of a single computer, which can be achieved by better circuit designs, smaller transistors in the CPU, a more sophisticated memory hierarchy, and so on. An alternative approach is to increase the throughput, which can be achieved by parallel processing, concurrent processing, or distributed processing. In fact, these processing paradigms apply to both software and hardware, and a computer network can be an implementation of the infrastructure.

The popularity of low-cost computer hardware and network infrastructure, in addition to other information appliances, makes ubiquitous computing practical. However, in order to use the pervasively available computing environment efficiently, different approaches to software development are necessary [5,14,17]. In this paper, we employ two optimization techniques, namely, particle swarm optimization (PSO) [13,21–23] and the genetic algorithm (GA) [9,12,24], as examples

* Corresponding author. Tel.: +886 918 952075.
E-mail addresses: 135170@mail.tku.edu.tw (F.-C. Chang), hch.nuk@gmail.com, huang.hc@gmail.com (H.-C. Huang).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.02.025

to demonstrate a means for increasing the utilization of the modern memory hierarchy. Both the PSO and GA are inherently different in concept and in implementation, and we can thus also make comparisons between them via simulations.

This paper is organized as follows. In Section 2, we review the software and hardware design characteristics of both the conventional and current approaches. We summarize the concepts of the exploitation of the memory hierarchy in Section 3. In Section 4, we analyze typical memory-access intensive applications, and propose several design guidelines for an algorithm design (or refactor) process. In Sections 5 and 6, we apply the guidelines to both particle swarm optimization and the genetic algorithm, respectively. Simulation results are provided and discussed in Section 7. Finally, we conclude this work in Section 8.

2. Trends in software and hardware developments

As we described in Section 1, the implementation of modern CPU design and network connectivity are far from their functional concepts, and we need to re-think software development paradigms. In this section, we briefly describe the general assumptions of the conventional development approaches. Then, we discuss some current development issues that are beyond the scope of conventional paradigms.

2.1. Conventional approach

In the past several decades, software and hardware development have been treated as almost separable issues. A programming language reflects one layer of the abstraction model between the hardware and the software [10]. The fundamental assumption is that abstraction can effectively decouple the correlation between adjacent layers. Thus, high-level software can utilize the low-level hardware transparently and efficiently.

This approach has worked for a long time. On the one hand, with the assumption of using a generic and simple computer architecture, software developers can focus on how to reduce the complexity of an algorithm. Hence, a well-designed algorithm is supposed to be efficient on a single processor with a unified memory model. The development of conventional methods thus focused on analyzing and designing efficient algorithms for sequential executions.

On the other hand, in hardware development, we assume that the software developer may be unaware of the hardware architecture. Well-designed hardware ought to execute the given instructions optimally. Therefore, different kinds of hardware design have been proposed. For instance, simultaneous electronic signal transmission is exploited to be parallel in circuits, and the locality of data accessing is exploited as part of the memory hierarchy design [19, pp. 468–491]. Because hardware circuits are parallel in nature, the hardware design languages, such as Verilog and VHDL, support parallelism directly.

2.2. Current trends and issues

As the dimension of a transistor becomes smaller, the circuit density on a chip becomes higher. The fabrication technology thus reaches the state where it is possible to integrate more than one CPU core in a package. That is, we have connectivity among several computation units inside one physical machine.

This enables distributed computing not only via network links but also via internal circuit connections. One of the differences between the two types lies in how a shared data object is accessed. When sharing a data object among networked processing units, the access time comprises the network transmission time and the memory-access time; when sharing a data object among circuited processing units, the access time is virtually the memory-access time. The network transmission efficiency can be bound by the network bandwidth. Similarly, the memory-access efficiency can be limited by the memory bus.

The multi-core design generates more challenges for the software design. It is obvious that a high-performance sequential algorithm may not be the optimal design to utilize the full hardware capability. It results from the overly simplified abstraction, which assumes generic and simple hardware support. Although there have been a few mature distributed algorithms [16], the number is not comparable to that of sequential algorithms [7]. One of the efficiency issues in distributed applications is data access. Data access is generally slower than a simple arithmetic operation, no matter whether the access is local or remote. Therefore, the performance bottleneck may be affected by data access patterns.

3. Issues for memory access

The data access pattern (or the software aspect) forms the traffic between the memory and the CPU (or the hardware aspect). Effective ways to improve the performance include either decreasing the number of data accesses or reducing the average memory-access clock cycles. Here, we focus on the latter strategy: to reduce the memory bus contention by carefully organizing the data structures and data access patterns in an algorithm. We discuss the concepts in this section.

3.1. Memory hierarchy

To develop a sequential algorithm, we often assume that there is a unified memory model (Fig. 1(a)). In a modern PC, the memory is organized by a hierarchical structure (Fig. 1(b)). The memory closer to the CPU core is smaller in size and faster in