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

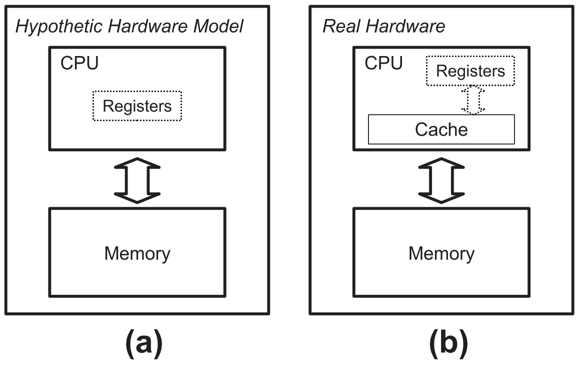

Fig. 1. The mismatch: (a) the memory model assumed by conventional algorithm designers; and (b) the hierarchical memory structure implemented in modern CPUs.

speed. Of course, there are coherence issues between adjacent hierarchical layers. There were many efficient and effective designs that were proposed and implemented. For example, the mapping among the cache lines and the memory blocks can be direct-mapped, set-associative, or fully associative [1,11]; the memory-to-cache updates can be affected by the replacement policies; the cache-to-memory updates can be write-through or write-back. Each of them can be implemented in more than one circuit design. No matter which scheme and design are chosen, the baseline is to produce the same results as those computed with the unified memory model. Therefore, in terms of software, the coherence problem can be safely omitted.

In recent years, due to various combinations of hardware and software integration, hardware/software co-design has become important in several fields. One of the techniques is to re-structure the algorithm to optimally utilize the available hardware resources. Inspired by this concept, we would like to focus on the effects of a cache on the software design. The concept of cache is based on the two locality properties, listed below.

Temporal locality: a data item tends to be accessed again within a short time.
Spatial locality: the neighboring items of a recently accessed data item also tend to be accessed within a short time.

In short, the two properties arise because computer programs are mainly designed to process regular, routine, and/or voluminous information. Statistical analysis of various kinds of applications reveals that most of the memory references follow the two properties, including the accesses to the instructions and to the data.

Based on the temporal and spatial localities, a cache is designed to be a fast memory device to hold a copy of the recently used main memory blocks. When a data reference occurs, a *hit* (found in the cache) indicates that the CPU can access the data directly in the fast memory device. In contrast, if a *miss* happens, the CPU should pay additional clock cycles to synchronize the data in the cache and in the main memory, which are called the *penalty cycles* and slow down the execution of the program. According to one investigation [2], the level-1 cache miss penalty is about 10 cycles in 86x family CPUs. In some cases, the penalty can be 30 cycles. If a software algorithm satisfies the locality properties (that is, when the cache hit rate is high), the cache increases the performance by decreasing the memory-access time. The worst case occurs when the data access pattern in the algorithm does not follow the locality properties. It causes data thrashing between the cache and the main memory, and introduces many cache miss penalty cycles.

3.2. Related research

Some hardware designs incorporate additional structures in the memory sub-system. One example is the DSP boards. They are specialized designs for high-performance signal processing, and they have sophisticated memory structures. For instance, the TI C64 board contains main memory (DRAM), local memory (SRAM), and cache. With this DSP board, a JPEG2000 encoder can be implemented as depicted in [15]. By carefully analyzing the locality properties in the codec, the data access pattern can be optimized as follows: (1) a to-be-processed data block should be manually loaded into the SRAM before being frequently accessed; and (2) the other difficult-to-determine or run-time-specific data accesses are optimized by the cache.

Without the prior knowledge of the cache structure, we can still develop an algorithm to utilize the cache efficiently at run-time. In 1999, an MIT master student completed his thesis on cache-oblivious algorithms [20]. Under the tall-cache structure assumption, a cache line size is not large. Therefore, an efficient algorithm is to process a small data chunk or a small number of data chunks as frequently as possible. A general approach is to use the divide-and-conquer design strategy. This strategy recursively divides the data items into smaller groups, until we can directly compute the results. The recursive division localizes the data access pattern to a small number of data items, and it never accesses them again. Even if the exact cache dimensions are unknown, the algorithm can still automatically satisfy the locality properties.

4. Proposed guidelines

Based on the previous studies [15,20], we have learned that localizing the data access pattern has the capability of improving the utilization of the cache in a modern computer. Hence, for pure (high-level) software design, the rule-of-thumb is to keep the maximal locality, so that the cache miss rate is minimized. In addition, due to modern cache design, cache lines are more or less set-associative. It is not likely for there to be many cache misses if only a few data items are frequently accessed in a given duration. Therefore, spatial locality has lower impact on the performance than temporal locality.

Based on the above discussion, a few design guidelines are proposed as follows:

Guideline #1: to exploit the temporal locality. If a data item (or a group of correlated data items) is accessed sparsely in the flow of the algorithm, we try to organize the access pattern to be contiguous operations. This guideline minimizes the number of cache replacements, and hence reduces the cache misses.

Guideline #2: to exploit the spatial locality. After processing a data item, we then process its neighboring items. If the data item is smaller than a cache line, its neighbors are probably loaded.

Guideline #3: to reorganize the data structures. Grouping semantically related data items in a structure is a commonly employed technique in program design. Ideally, when they are packed in a memory block, the number of cache lines required to hold the necessary data is minimized. Sometimes this approach inherently satisfies both the temporal and spatial locality. However, semantically related items may not be processed in contiguous operations. We may need to break a "big" data structure into several "small" structures according to the access patterns and may sacrifice the intuitive structure of a data object. Each small structure is accessed as frequently as possible in contiguous operations. Otherwise, either temporal locality or spatial locality is broken.

If an algorithm can be represented in map-reduce style, an alternative analysis can be derived. As defined in [4], the `map` operator and the `reduce` operator are employed to process a list with the given function:

$$\begin{aligned}&\mathrm{map}\,f\ [e_1, e_2, e_3, \ldots, e_n] \rightarrow [f(e_1), f(e_2), f(e_3), \ldots, f(e_n)]\\&\mathrm{reduce}\,g\ []\ [e_1, e_2, e_3, \ldots, e_n] \rightarrow g(e_1, g(e_2, g(e_3, \ldots g(e_n, []))))\end{aligned}$$

The two operators iterate through all the elements of the given list. The difference between the two is that the `map` operator processes each element independently. Suppose there are two sequential `map` operations: one applies the function $f_1(\cdot)$ and the other applies the function $f_2(\cdot)$. If the two `map`s are cascaded, they can be merged as one `map` operator with the combined function $f_2(f_1(\cdot))$. That is,

$$\mathrm{map}\,f_2\ (\mathrm{map}\,f_1\ [e_1, e_2, e_3, \ldots, e_n]) = \mathrm{map}\,f_2(f_1(\cdot))\ [e_1, e_2, e_3, \ldots, e_n]$$

This combination is similar to making the sparse operations contiguous.

There exists a less straightforward transformation that also makes sparse operations contiguous. Let $f \circ g(\cdot)$ be the direct composition of the two functions $f(\cdot)$ and $g(\cdot)$, i.e., $f \circ g$ returns a pair of values:

$$\begin{aligned}&f \circ g(e, m) = (f(e), g(m))\\&f \circ g(e) = (f(e), g(e)), \quad \text{when } e \text{ is identical to } m.\end{aligned}$$

Assume that there are two related lists of the same size, so the two `map` operations can be combined as follows:

$$\begin{aligned}&\left\{\begin{array}{l}\mathrm{map}\,f\ [e_1, e_2, e_3, \ldots, e_n]\\ \mathrm{map}\,g\ [m_1, m_2, m_3, \ldots, m_n]\end{array}\right\}\\&= \mathrm{map}\,f \circ g\ [(e_1, m_1), (e_2, m_2), (e_3, m_3), \ldots, (e_n, m_n)]\end{aligned}$$

This transformation suggests that we may consider a data structure containing the elements e and m as the members. A special case of the above transformation is that the list $\{e_i\}$ is identical to the list $\{m_i\}$. No new data structure is needed, and the transformation makes the sparse operations contiguous.

The `reduce` operator sequentially processes the list elements, and it satisfies the spatial locality inherently. Moreover, in practical designs, it usually represents a dependency boundary. That is, the statements between two `reduce` operators form a code section. In a code section, the data items may be internally dependent but externally independent. If two `reduce` operations apply to the same list, we may consider the following transformation:

$$\begin{aligned}&\left\{\begin{array}{l}\mathrm{reduce}\,f\ []\ [e_1, e_2, e_3, \ldots, e_n]\\ \mathrm{reduce}\,g\ []\ [e_1, e_2, e_3, \ldots, e_n]\end{array}\right\}\\&= \mathrm{reduce}\,f \circ g\ []\ [e_1, e_2, e_3, \ldots, e_n]\end{aligned}$$

In summary, with the aforementioned transformation guidelines, we can make the following observations. No matter what programming paradigm is used to express the algorithm, we can refactor the algorithm to efficiently utilize the cache as long as the localities are carefully identified and analyzed.

5. Application to particle swarm optimization

In this section, we demonstrate how to apply the proposed guidelines to a simple PSO algorithm. In PSO, each particle stands for a solution vector. A fitness value is associated with the solution, according to the given fitness function. A particle moves in the solution space and forms a trace. For each particle, we record both the current solution, and the best solution along its trace. A global-best solution is also recorded. In some PSO variations, we also record the best solution reported by the neighbors. The local-best, global-best, and neighborhood-best particles are the referential attraction points that make the particles have the tendency to move toward the best position.

The movement of a particle is controlled by the velocity. It is updated by a linear combination of the current particle velocity and the velocities to the attraction positions. The weighting factor for the current particle velocity is a constant number, representing the momentum. The velocities to the attraction positions are weighted by the random vectors, which reduce the probability of being trapped in to a local minimum and make a particle explore the solution space while moving toward the candidate of the best solution.

5.1. A basic PSO algorithm

Because PSO has been developed for a few years, there are many variations in comparison with the original version of PSO [13]. To demonstrate our optimization concept, we chose a basic PSO algorithm as the example. Suppose that a solution can be represented by an M-dimensional vector, and there are N particles. The goal is to minimize the fitness function, with notations as follows:

(1) The particle is indexed by i, and the dimension is indexed by j.
(2) The particle positions are $\{x_i\}$, and their associated velocities are $\{v_i\}$.
(3) The local-best locations are $\{\hat{x}_i\}$.
(4) The global-best location is $\hat{g}$.
(5) The fitness function is $f(\cdot)$ (the smaller the value is, the better the fitness becomes).
(6) $\{r1_i\}$ and $\{r2_i\}$ are random vectors, where $r1_{ij} \in [0.0, 1.0]$ and $r2_{ij} \in [0.0, 1.0]$.
(7) ω, c_1, and c_2 are the control parameters specified by the user.
(8) The binary operator $\circ$ means the element-to-element multiplication of the two vectors.

The PSO algorithm can be briefly described as follows:

```
Step 1. Initialize the $\{x_i\}$ as random vectors in the solution space.
Step 2. Initialize the $\{v_i\}$ as zero vectors.
Step 3. Initialize $\{\hat{x}_i | \hat{x}_i \leftarrow x_i\}$.
Step 4. $\hat{g} \leftarrow \arg\min_{\hat{x}_i} f(\hat{x}_i)$.
Step 5. Stop the process if the global-best particle satisfies the termination condition.
Step 6. Create random vectors $\{r1_i\}$ and $\{r2_i\}$.
Step 7. Update velocities $\{v_i | v_i \leftarrow \omega v_i + c_1 r1_i \circ (\hat{x}_i - x_i) + c_2 r2_i \circ (\hat{g} - x_i)\}$.
Step 8. Update positions $\{x_i | x_i \leftarrow x_i + v_i\}$.
Step 9. Update local-best positions $\{\hat{x}_i | \hat{x}_i \leftarrow x_i, \text{if} f(x_i) < f(\hat{x}_i)\}$.
Step 10. Update global-best position $\hat{g} \leftarrow \text{argmin}_{\hat{x}_i} f(\hat{x}_i)$.
Step 11. Go to Step 5.
```

It can be easily shown that each particle is an independent processing unit. The related information includes the position x_i, the velocity v_i, and the local-best position $\hat{x}_i$. We can also observe that they are dependently processed at Steps 1–3 and Steps 7–9. Therefore, the straightforward method to increase the locality is to group $(x_i, v_i, \hat{x}_i)$ as a structure p_i.

Next, we can observe Step 4 and Step 10. The two steps iterate through all the available $\hat{x}_i$ to find the global-best position. At Step 3, we have already accessed all the $\hat{x}_i$'s. This implies that the operations applied to $\hat{x}_i$'s in Step 3 and Step 4 can be combined together to increase the temporal locality. Similarly, the corresponding operations of Step 9 and Step 10 can also be combined.

We can also observe that Step 6 through Step 10 are dependent. The $r1_i, r2_i, v_i, x_i$, and $\hat{x}_i$ parameters are accessed with dependencies. If we simply follow the algorithm, the data items for the ith particle are accessed sparsely during the execution of the steps. Because the algorithm updates each particle independently, we can combine the operations from Step 6 to Step 10 to process the particles one after another.

5.2. The rearranged PSO algorithm

According to the analysis in the previous section, we can reorganize the basic PSO algorithm by grouping the data in the particle structure and by combining the processing operations. In the following revised version, we use p_i to represent the ith particle; $p_i \cdot x$, $p_i \cdot v$ and $p_i \cdot \hat{x}$ mean the current position, the velocity, and the local-best position, respectively; r_1 and r_2 are the random weighting vectors for updating the particle velocity.

The rearranged PSO algorithm can be briefly described as follows.

```
Step 1. Initialize: For each particle $p_i$
    Step 1.1. $p_i \cdot x$ is a random vector.
    Step 1.2. $p_i \cdot v$ is a zero vector.
    Step 1.3. $p_i \cdot \hat{x} \leftarrow p_i \cdot x$.
    Step 1.4. $\hat{g} \leftarrow p_i \cdot \hat{x}$, if $f(p_i \cdot \hat{x}) < f(\hat{g})$.
Step 2. Stop the process if the global-best particle satisfies the termination condition.
Step 3. Search: For each particle $p_i$
    Step 3.1. Create random vectors $r_1$ and $r_2$.
    Step 3.2. $p_i \cdot v \leftarrow \omega p_i \cdot v + c_1 r_1 \circ (p_i \cdot \hat{x} - p_i \cdot x) + c_2 r_2 \circ (\hat{g} - p_i \cdot x)$.
    Step 3.3. $p_i \cdot x \leftarrow p_i \cdot x + p_i \cdot v$.
    Step 3.4. If $f(p_i \cdot x) < f(p_i \cdot \hat{x})$
        Step 3.4.1. $p_i \cdot \hat{x} \leftarrow p_i \cdot x$.
        Step 3.4.2. $\hat{g} \leftarrow p_i \cdot \hat{x}$, if $f(p_i \cdot \hat{x}) < f(\hat{g})$.
Step 4. Go to Step 2.
```

In the rearranged algorithm above, both data item grouping and operation combining should be applied. Data item grouping may increase the spatial locality, and operation combining may enhance the temporal locality. As we will see in Section 7.1, when only data item grouping is applied, the performance is degraded.

6. Application to the genetic algorithm

Evolutionary algorithms are population-based or individual-based. There is potential parallelism in the processing of the individuals, which implies that we may organize an evolutionary algorithm to exploit the localities based on the access pattern to the individuals. In addition to PSO, the genetic algorithm (GA) is another popular technique for this operation. In this section, we analyze the conventional simple GA, identify the related localities, and then apply the proposed guidelines to revise the algorithm.

6.1. Conventional genetic algorithm

In most of the literature, a genetic algorithm is described as several processing stages:

```
Step 1. Generate the initial population.
Step 2. Evaluate the fitness of each chromosome.
Step 3. Terminate the execution if the termination condition is satisfied.
Step 4. Alternate the population.
Step 5. Apply the genetic operations, such as cross-over and mutation.
Step 6. Go to Step 2.
```

Genetic algorithm is suitable for parallel processing. There are operations that can be executed without data dependencies. For example, the evaluation process in Step 2 can be parallelized because each chromosome is independent of the others. If we have only one processor, the above steps cause inefficiency of memory access.

The problem is that the temporal locality can only be well preserved with difficulty. In Step 4 and Step 5, we evolve the old population to produce the new one. Next, we iterate through the population again to compute the fitness of each chromosome. If the population size is large, one chromosome is generated (in Step 4 and Step 5) and temporarily stored in the cache, written back to the main memory due to the subsequently generated chromosomes, and then loaded into the cache again while evaluating the fitness value. Because a cache miss results in an extra data fetching procedure, the performance is degraded by the penalty cycles.

6.2. The reorganized GA

The above problem comes from the decreased temporal locality. It can be resolved by reordering and reorganizing the operations applied to a chromosome. The rearranged GA can be described as follows:

```
Step 1. Generate the initial population. *For each chromosome, evaluate the fitness right after it is generated.*
Step 2. Terminate the execution if the termination condition is satisfied.
Step 3. Alternate the population.
```

Step 4. Apply the genetic operations, including cross-over, mutation, and reproduction. *For each chromosome, evaluate the fitness right after it is generated.*
Step 5. Go to Step 2.

In comparison with the conventional GA, we evaluate the fitness value right after a chromosome is generated. The revised GA accesses each chromosome as soon and frequently as possible. Therefore, it preserves more temporal locality than the original one does. Based on the characteristics and operations of GA, although this scheme does not modify as much as what we did to the PSO, it can effectively reduce the cache miss rate, as shown in Section 7.2.

7. Simulations and discussion

To show the effectiveness of the revised PSO and GA algorithms, we conducted a few simulations and discuss the results in this section. In Section 7.1, the PSO algorithms mentioned in Section 5 were used. We also simulated a poor revision to depict the deficiency of an improper algorithm reorganization. In Section 7.2, the GA algorithms mentioned in Section 6 are simulated and discussed. Additional discussions about the effects of the guidelines are described in Section 7.3.

Note that we make two assumptions for the following simulations. First, in an evolutionary algorithm, the major memory references are the iterative accesses to the population. The number of memory references in processing a solution object depends on the fitness function and the population alternation. Because we would like to show the locality of an individual solution object, the simulation programs are designed to keep the processing of a solution object as simple as possible. To achieve this goal, we use statically allocated arrays for solutions, incorporate simple fitness functions, and use simple evolution strategies. Second, we assume that the processing flow is almost identical in each iteration. Therefore, the convergence of the solution will slightly affect the cache miss rate. We simply simulated a fixed number of evolutions. The number of evolutions should be large enough for two reasons: the first is to show a reasonable number of cache misses; and the second is to make the number of memory accesses during initialization negligible. In addition, the user-supplied parameters were assigned reasonable constant values in the same series of simulations.

Based on some preliminary experiments, we found that the example PSO and GA programs made sensible numbers of level-1 data cache (D1 cache) misses. The level-1 instruction cache did not have many misses because each program was small enough to be stored in the level-1 instruction cache. In addition, the memory consumption of each program was not large enough to trigger a sensible number of level-2 cache misses. Therefore, we concentrated on the D1 cache misses. In the following simulations, we varied the population size and compared the numbers of D1 cache misses. We used the memory profiling program *valgrind* [18] with its built-in tool *cachegrind* to simulate the memory/cache behavior. The cache was configured as 32 kB 8-way set-associative with a 64-B cache line.

7.1. Simulation results for PSO

In the simple PSO program, five quantities need to be recorded into five arrays, including (1) the current particle locations, (2) the current fitness values, (3) the velocities, (4) the local-best locations, and (5) the local-best fitness value. In the 10-dimensional search space, let $X = (x_1, x_2, x_3, \ldots, x_{10})$ represent a particle location. The fitness function is specified as

$$f(X) = \sum_{i=1}^{10} (x_i - 0.11 \times i)^2$$

In the PSO simulations, we specified $\omega = 0.8, C_1 = C_2 = 2$ and performed 1000 iterations to collect the statistical information.

Table 1 lists the simulation results over different particle numbers. For each case, the two versions of PSO were simulated and compared. The number of data references (D refs) and the number of D1 cache read/write misses (D1 misses) were recorded. We could see that the number of D refs and the number of D1 misses increased as the number of particles grew. While the D refs were roughly the same in both versions, the D1 misses were quite different.

Fig. 2 demonstrates the D1 cache miss rate curves of the original and the revised PSO algorithms. When the particle number is less than 200, the cache miss rates are almost the same because all the particle information can be loaded into the cache. In the simulation program, a particle was associated with three 10-dimensional vectors and two scalar numbers. In other words, a particle occupied 256 bytes. Because the cache is 32 kB, at most 128 particles could stay in the cache.

When the number of particles was large enough to trigger the replacement policy frequently, the cache miss rate became saturated. Comparing the two curves in Fig. 2, the miss rate of the revised algorithm is about 27% less than the original one. This implies that the revised version preserves more locality, as expected.

As mentioned in Section 5.2, the example PSO algorithm should be reorganized by considering both types of locality properties. If we simply group the particle information in a flat structure and create an array to hold all the particles, the spatial locality becomes degraded in comparison with that of the original algorithm, which increases the miss rate by 45%, as shown in Fig. 3. From this example, we learn that a poorly analyzed and/or refactored algorithm can behave inefficiently at runtime.

Table 1
Simulation results of PSO.

Particles	Original PSO		Revised PSO	
	D1 misses	D refs	D1 misses	D refs
10	6.72E+03	1.77E+07	6.72E+03	1.76E+07
20	6.77E+03	3.48E+07	6.78E+03	3.46E+07
30	6.81E+03	5.19E+07	6.81E+03	5.16E+07
40	6.86E+03	6.90E+07	6.86E+03	6.86E+07
50	6.91E+03	8.61E+07	6.91E+03	8.56E+07
60	6.95E+03	1.03E+08	6.95E+03	1.03E+08
70	7.00E+03	1.20E+08	7.00E+03	1.20E+08
80	7.05E+03	1.37E+08	7.06E+03	1.37E+08
90	7.10E+03	1.55E+08	7.10E+03	1.54E+08
100	7.16E+03	1.72E+08	7.17E+03	1.71E+08
200	8.04E+05	3.43E+08	8.06E+05	3.41E+08
300	1.48E+06	5.14E+08	1.21E+06	5.11E+08
400	2.12E+06	6.85E+08	1.61E+06	6.81E+08
500	2.67E+06	8.56E+08	2.01E+06	8.51E+08
600	3.21E+06	1.03E+09	2.41E+06	1.02E+09
700	3.75E+06	1.20E+09	2.81E+06	1.19E+09
800	4.28E+06	1.37E+09	3.21E+06	1.36E+09
900	4.83E+06	1.54E+09	3.61E+06	1.53E+09
1000	5.37E+06	1.71E+09	4.01E+06	1.70E+09

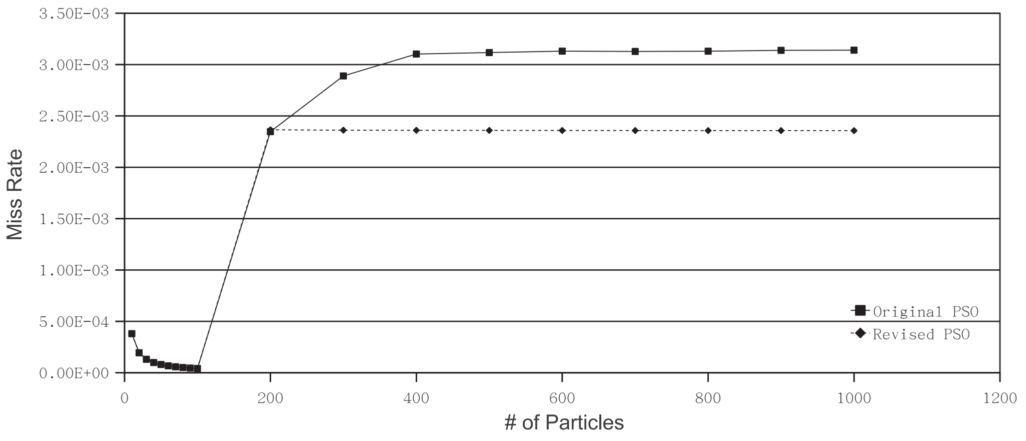

Fig. 2. The D1 cache miss rates of the original and the revised PSO.

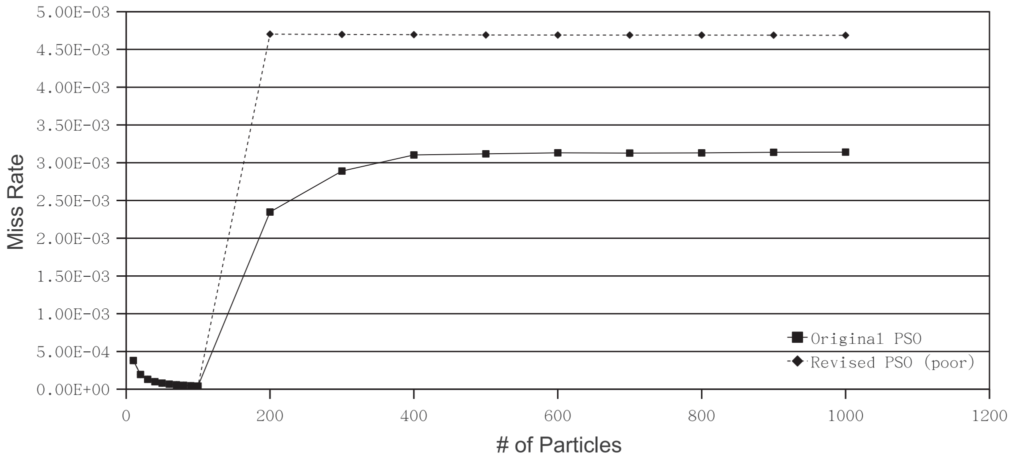

Fig. 3. The D1 cache miss rates of the original and the poorly revised PSO.

7.2. Simulation results for GA

To test the effectiveness of the revised genetic algorithm, we refer to a simple GA program [24] and make the necessary modifications. The program follows the flow of the conventional GA and allocates a two dimensional array to hold all the chromosomes. Each chromosome consumes 7 bytes to represent the true(1)/false(0) states of 7 objects. The fitness function is defined to select the solution “1001011”.

Table 2 lists the simulation results over different population sizes. For each population size, the two versions of GA were simulated. Similar to the results of the PSO, the number of D refs and the number of D1 misses increased as the population size grew. While the D refs were roughly the same in both versions, the D1 misses were quite different.

Fig. 4 depicts the curves of the cache miss rate to the population size. Because each chromosome takes 7 bytes, we may expect that the 32 kB cache can hold about 4600 chromosomes. The miss rate curves show that the locality effect is not obvious when the population size is less than 5000. After increasing the population size to more than 6000, the cache miss rate became saturated. Comparing the two curves, the miss rate of the revised algorithm is a half of that of the original one. It implies that the revised version preserves more locality, as expected.

7.3. Discussions of simulation results

Current compilers are often equipped with one or several code optimization modules. The well-known optimization techniques, such as loop unrolling and common expression elimination, are implemented. The optimizers can only follow the given algorithm and data structure to analyze the *static* characteristics. The locality property is a *run-time* characteristic, and the common optimizers cannot easily deal with this aspect.

Table 3 and Fig. 5 demonstrate the additional simulation results for PSO. In this experiment, we turned on the optimization (GCC -O2) when compiling the PSO programs. We could see that the optimizer reduced the *memory references* in both versions. In addition, the cache miss rate was reduced by about 22% after applying the guidelines to revise the original algorithm.

In both kinds of the algorithms, the miss rate drops when the population size is small. The major reason is that the cache can hold all the solution objects. Once a solution object is first accessed, the object and its neighbors are loaded into the cache

Table 2
Simulation results of GA.

Population	Original GA		Revised GA	
	D1 misses	D refs	D1 misses	D refs
1000	6.78E+03	9.87E+07	6.78E+03	9.84E+07
2000	6.91E+03	1.97E+08	6.91E+03	1.96E+08
3000	7.09E+03	2.95E+08	7.09E+03	2.94E+08
4000	1.10E+04	3.93E+08	1.06E+04	3.92E+08
5000	9.49E+04	4.91E+08	5.68E+04	4.90E+08
6000	1.43E+05	5.89E+08	7.59E+04	5.88E+08
7000	1.64E+05	6.87E+08	8.69E+04	6.85E+08
8000	1.86E+05	7.85E+08	9.79E+04	7.83E+08
9000	2.08E+05	8.83E+08	1.09E+05	8.81E+08
10,000	2.30E+05	9.82E+08	1.20E+05	9.79E+08
11,000	2.52E+05	1.08E+09	1.31E+05	1.08E+09
12,000	2.74E+05	1.18E+09	1.42E+05	1.17E+09

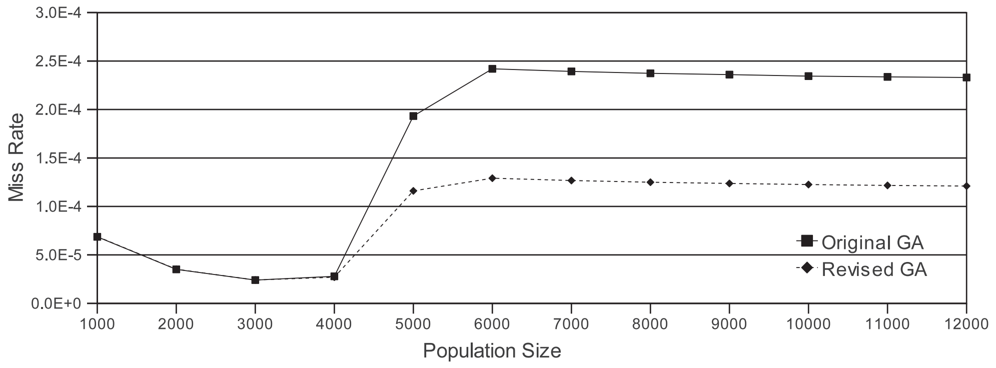

Fig. 4. The D1 cache miss rates of the original and the revised GA.

Table 3
Simulation results of compiler-optimized PSO.

Particles	Original PSO		Revised PSO	
	D1 misses	D refs	D1 misses	D refs
10	6.73E+03	1.46E+07	6.72E+03	1.40E+07
20	6.77E+03	2.86E+07	6.76E+03	2.75E+07
30	6.81E+03	4.26E+07	6.81E+03	4.10E+07
40	6.86E+03	5.62E+07	6.86E+03	5.44E+07
50	6.91E+03	7.01E+07	6.90E+03	6.79E+07
60	6.95E+03	8.40E+07	6.95E+03	8.13E+07
70	7.00E+03	9.79E+07	7.00E+03	9.48E+07
80	7.05E+03	1.12E+08	7.05E+03	1.08E+08
90	7.10E+03	1.26E+08	7.10E+03	1.22E+08
100	7.16E+03	1.40E+08	7.15E+03	1.35E+08
200	8.01E+05	2.79E+08	8.06E+05	2.70E+08
300	1.49E+06	4.18E+08	1.21E+06	4.04E+08
400	2.12E+06	5.57E+08	1.61E+06	5.39E+08
500	2.66E+06	6.96E+08	2.01E+06	6.74E+08
600	3.21E+06	8.35E+08	2.41E+06	8.08E+08
700	3.75E+06	9.74E+08	2.81E+06	9.43E+08
800	4.28E+06	1.11E+09	3.21E+06	1.08E+09
900	4.83E+06	1.25E+09	3.61E+06	1.21E+09
1000	5.38E+06	1.39E+09	4.01E+06	1.35E+09

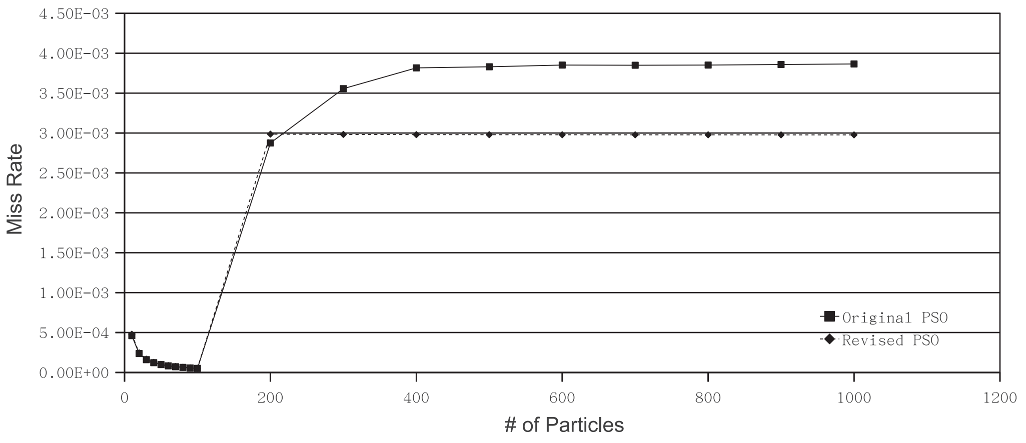

Fig. 5. The D1 cache miss rates of the original and the revised PSO with compiler optimization.

and can never be missed again. The number of total misses are composed of (1) the first-time access to the solution objects and (2) other run-time implicit memory accesses. The overall effect is that the number of misses increases more slowly than the number of total data references does, and hence the miss rate curve drops.

Our simulations demonstrate a carefully analyzed and reorganized algorithm that has lower cache miss rate. Consequently, we can reduce the costly cache miss penalty cycles. Considering practical applications, two facts give this approach limited improvements for common programs:

- The compiler optimizes the number of memory accesses, which is more effective in saving CPU cycles.
- A modern computer often has a large enough cache device, which makes the original miss rate quite low. The reduction of cache misses is thus somewhat unremarkable.

Although it may have limited effects on common programs, it would be useful for large-volume and distributed data processing. In the former type of application, we process a large amount of data. If we can keep the cache miss rate as low as possible, the overall effect would be non-trivial. Suppose we have a busy multi-tasking environment. A properly reorganized program can reduce the memory bus thrashing (fewer cache misses). Because the CPU is not the only component that accesses the memory, it implies that the other hardware controllers have more chances to access the memory. This may shorten the execution time of the other programs. The overall effect is that the throughput is increased.

In the latter type of application (distributed data processing), data items are distributed over the network nodes. Because network communication is costly, we have to reduce the amount of data transfer. We can adopt the guidelines to analyze the localities in a distributed algorithm, localize the processing operations to a small set of data items and reduce the run-time data transfer.

8. Conclusions

In this paper, we reviewed the fundamental assumptions of a cache. With the cache-aware and cache-oblivious concepts, we learned that an algorithm can be rearranged to utilize the cache efficiently. Because the conventional algorithm design is based on an overly simplified abstraction model, the result may cause unnecessary cache miss penalties. In order to preserve the temporal and spatial locality in an algorithm, our proposed strategy is to reorganize the designed algorithm so that a data element is accessed as soon and as frequently as possible.

Depending on the analysis, we can rearrange a given algorithm and/or the data structure by temporal locality-based, spatial locality-based, or combined locality-based approaches. Due to the limited capability of the compiler optimization, our proposed guidelines still work even when the optimization is turned on. The concept of locality-based reorganization can be extended to distributed algorithms to keep the minimum data transfer among network nodes.

To depict the feasibility of the proposed guidelines, we applied them to two example programs. Both the PSO and GA programs were simulated for different population sizes. The results demonstrate that the cache miss rate is greatly reduced in the refactored version, which implies that our proposed guidelines are potentially useful for a memory-access intensive program.

Acknowledgement

This work is partially supported by National Science Council (Taiwan) under Grant No. NSC 98-2218-E-032-008.

References

[1] A. Agarwal, S.D. Pudar, Column-associative caches: a technique for reducing the miss rate of direct-mapped caches, in: Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'20), San Diego, California, 1993, pp. 179–190.
[2] V. Babka, P. Tůma, Investigating cache parameters of ×86 family processors, Computer Performance Evaluation and Benchmarking 5419 (2009) 77–96.
[3] A.W. Burks, H.H. Goldstine, J. von Neumann, Preliminary discussion of the logical design of an electronic computing instrument, in: W. Aspray, A. Burks (Eds.), Papers of John von Neumann, MIT Press, 1987, pp. 97–146.
[4] F.-C. Chang, H.-C. Huang, A programming model for distributed content-based image retrieval, in: IEEE International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2007), Kaohsiung, Taiwan, 2007, pp. II–37–II–40.
[5] H. Choo, Y. Lee, S.M. Yoo, DIG: degree of inter-reference gap for a dynamic buffer cache management, Information Sciences 176 (8) (2006) 1032–1044.
[6] D.W. Clark, Pipelining and performance in VAX 8800 processor, in: Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM, Palo Alto, Californoia, 1987, pp. 173–177.
[7] T.H. Cormen, Introduction to Algorithms, MIT Press, 2001.
[8] J. Fisher, S. Freudenberger, Predicting conditional branch directions from previous runs of a program, in: Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM, Boston, 1992, pp. 85–95.
[9] D.B. Fogel, Evolutionary Computation, IEEE Press, New York, 1998.
[10] K. Irvine, Assembly Language for Intel-based Computers, fifth ed., Prentice-Hall, 2006.
[11] N.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, Seattle, 1990, pp. 364–373.
[12] M. Kantardzic, Data Mining: Concepts, Models, Methods, and Algorithms, IEEE Press, 2003.
[13] J. Kennedy, The particle swarm: social adaptation of knowledge, in: Proceedings of the IEEE International Conference on Evolutionary Computation, 1997, pp. 303–308.
[14] Y. Leu, J. Hung, An energy efficient re-access scheme for data caching in data broadcast of a mobile computing environment, Information Sciences 177 (24) (2007) 5538–5557.
[15] C.-C. Liu, Acceleration and Implementation of JPEG2000 Encoder on TI DSP Platform, Master's Thesis, Dept. of Electronics Engineering, National Chiao Tung University, Taiwan, 2006.
[16] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, Inc., 1997.
[17] Y. Na, C. Leem, I. Ko, ACASH: an adaptive web caching method based on the heterogeneity of web object and reference characteristics, Information Sciences 176 (12) (2006) 1695–1711.
[18] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), 2007, pp. 89–100.
[19] D.A. Patterson, J.L. Hennessy, Computer Organization and Design, third ed., Elsvier Inc., San Francisco, 2005.
[20] H. Prokop, Cache-oblivious Algorithms, Master's Thesis, MIT, 1999.
[21] P.N. Suganthan, Particle swarm optimiser with neighbourhood operator, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 1999), 1999, pp. 1958–1962.
[22] P.K. Tripathi, S. Bandyopadhyay, S. Pal, Multi-objective particle swarm optimization with time variant inertia and acceleration coefficients, Information Sciences 177 (22) (2007) 5033–5049.
[23] F. van den Bergh, A. Engelbrecht, A study of particle swarm optimization particle trajectories, Information Sciences 176 (8) (2006) 937–971.
[24] D.V. Vargas, Hello World in Genetic Algorithm, 2007. <http://fog.neopages.org/helloworldgeneticalgorithms.php>