Improving Paging Performance With Object Prefetching

Neelam Saboo
Laxmikant V. Kalé
{saboo, kale}@cs.uiuc.edu

Dept. of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801

Abstract

Many computationally intensive parallel programs are also memory intensive. Even when the memory required to model a particular application is larger than the available memory, the virtual memory system permits the program to run. However, this comes at a substantial performance cost because disk accesses incurred by paging are substantially slower than memory accesses. Out-of-core techniques are often developed for each application separately to deal with this issue. We present a generic, application independent, technique that automatically improves paging performance, by using prefetching and multithreading. We take advantage of the virtualization provided by Charm++ objects in parallel programs. The data driven objects provide the prediction mechanism necessary for an effective prefetching scheme. An analytical model as well as experimental data is presented to demonstrate the advantages of the proposed scheme.

1 Introduction

Parallel programs often deal with large problems which are computationally intensive and have huge memory requirements. In spite of the relatively large amount of memory available in today's computer systems, there are many parallel applications that require more memory than available. Virtual Memory mechanism satisfies the memory requirement of such applications by paging the required pages of memory from physical memory. However, page faults take several milliseconds compared to few nanoseconds for memory accesses, leading to some performance loss. Therefore, reducing or hiding page faults is crucial in achieving high performance. We aim at improving performance for such programs by exploring and analyzing the concept of *Object Prefetching*.

If the data referenced by an application is not in main memory, a page fault occurs and the page is fetched from secondary memory to main memory. CPU is essentially idle during

that period. *Software prefetching* is a technique for tolerating memory latency by explicitly getting data in memory before it is actually needed. Thus, when the program asks for the page, the page is likely to be available instantaneously instead of going through a page fault.

The paper explores a multi-threading approach for data prefetching. It creates one or more threads that will fetch the data in memory before it is actually required. When one thread is blocked on paging, system passes control to other threads [1]. Thus the performance loss due to paging can be reduced by overlapping it with computation.

Experiments are conducted on Solaris and Linux platforms using POSIX-threads to study the behavior of paging with different parameters.

This approach presents a major challenge. To fetch the required data, program should correctly predict what data is going to be accessed in near future. Data-driven object paradigms overcome this challenge. In such systems, the execution of objects' method is triggered by availability of messages (method invocations) under the control of a prioritized scheduler. Thus the scheduler has the knowledge of objects that are going to receive messages in near future. In the following section, we discuss one such data-driven object oriented paradigm.

2 Data-driven Object Paradigms: CHARM++

CHARM++ is an object oriented parallel language [2]. It is different from traditional parallel programming paradigms such as message passing and shared variable programming in that its execution model is *message-driven*. The computations in CHARM++ are triggered on arrival of associated messages. These computations in turn trigger more messages to other processors that might cause more computations on those processors. CHARM++ is based on a concept of parallel objects called *Chares*. Chares can communicate with each other only through messages. CHARM++ also supports notion of a *Group* and *Array*. A Group has one representative on each processor. Arrays are arbitrarily sized collection of chares. Each array element can send and receive messages just like chares, and computation is performed

upon arrival of those messages.

Each chare contain a number of *entry methods*, which are methods that can be invoked from remote processors. CHARM++ provides system calls to asynchronously create remote chares and to asynchronously invoke entry methods on remote chares by sending messages to those chares. This asynchronous message passing is the basic interprocess communication mechanism in CHARM++. CHARM++ also allows application to associate priorities with these messages, so that high priority messages will be handled before lower priority messages. CHARM++ is built on top of the Converse run-time framework. Converse provides portable, efficient implementations of all the functions typically needed by a parallel language of library. For example, Converse provides an architecture-independent interface to most thread functions, thread scheduling, synchronization of variables, and message passing. The CHARM++ environment can run on combinations of many different operating systems, including Linux, Irix, Solaris, Windows NT as well as many specialized parallel computers like Cray T3E, IBM's SP, etc.

2.1 Scheduler Overview

Each processor is running a Converse scheduler [3], which is responsible for all message reception and scheduling. Each message contains a function pointer and some bytes of user data . The scheduler's job is to repeatedly pick and process the messages in the order of their priority. Messages need to register with converse environment informing it about the association of handlers with messages. Processing a message consists of calling the handler which is encapsulated in the message, passing in the message's data.

Whenever a message is sent from a chare or an array element to another chare or an array element, the message is inserted in the schedulers' queue which is a prioritized queue. The *scheduler thread* runs a loop which retrieves messages from a prioritized queue and pass it to a message handler function.

Thus, the scheduler has the knowledge of messages which are going to be executed in

```
while(true)
{
  dequeue a message from csdSchedQueue ;
  if(message is present)
    handle the message.
  else
    loop idle until a message is received.
}
```

near future and therefore does not need any prediction mechanisms for prefetching.

3 Object Prefetching

In this section we study the feasibility of the concept of prefetching and provide theoretical support for it. We write a simple benchmark that models the behavior of a data-driven program which uses a high amount of memory and accesses the data so that it causes significant amount of paging. The performance of the application is then measured in terms of access time per object, which is averaged over total number of objects. Experiments are performed on Solaris and Linux with different scenarios: in absence of paging, with paging, with prefetching, different computation times.

3.1 Benchmark Application

The benchmark program consists of a dynamically allocated array of objects (N), each object of size of a page, 4K (S). The program creates a *Computation Thread*, which selects an object at a time and performs some computation on it, in a loop for some fixed number of iterations. Objects are accessed randomly so that consecutive accesses are on different pages. Number of iterations is 10 times the number of objects to avoid the effect of *cold misses* due to initial few access. Effective time per object is measured as the total time divided by the number of iterations.

A lock-free Producer-Consumer queue is maintained which holds the indices of the objects in the array. *Prefetch thread* accesses an object in the array and enqueues the index in the

queue. The computation thread dequeues an index from the queue and performs computation on that object. Prefetch thread maintains a leash such that it will prefetch only when the queue size is less than the leash size. This is a programmable parameter to study the effect of different leash sizes on prefetching. To simplify the benchmark application, prefetch thread decides what data is going to be accessed in near future instead of computation thread governing it.

The idea behind this approach is that as prefetch thread accesses the objects, page faults will be taken by it. As computation thread comes to a point to access that object, it is already brought in memory. So it will not incur a page fault, assuming the object is not paged out by this time. Keeping leash to some reasonable size such as 10-30, we can statistically assume that pages will not be thrown out between the time prefetch thread brings it into memory and computation thread tries to access it. If LRU page replacement policy is used by the operating system, it becomes even less likely. By maintaining the leash, there is some work queued up for computation thread. As prefetch thread is taking a page fault, operating system will pass the control to other threads. computation thread thus can keep working on the objects brought in memory earlier. If these two times overlap for a maximum time, the program performs as though there is no paging. When the queue is filled up to leash size, prefetch thread yields to computation thread. Also, if queue is empty, computation thread yields to prefetch thread. Code for prefetch thread and computation thread is in Following figure.

The benchmark application is similar to CHARM++ scheduler we discussed in the previous section as prefetch thread knows what data the application needs in near future. Thus, the same idea of prefetching can be incorporated in CHARM++ scheduler. A prefetch thread can be introduced in CHARM++ scheduler which will access the schedulers queue and prefetch the objects which are going to receive message in near future, while computation thread only handles the messages and performs computation.

```
void* computationThread (void* info)
{
  while(countW < ITER)   {
    if(PCQueueEmpty(bufQ)) {
      sched_yield();
      continue;
    }
    index = PCQueuePop(bufQ) ;
    for(i =0 ; i < 100; i++) {
      int rand = random() % N ;
      sum += objs[index]->A[rand] ;
    }
    sum = doWork(sum) ;
    countW++ ;
  }
}
```

```
void* prefetchThread (void *info)
{
  while( (countP) < ITER)   {
    if(PCQueueLength(bufQ) >= leash) {
      sched_yield();
      continue ;
    }
    randIndex = random() % S ;
    for(i =0, rand = 0 ; i < 5; i++) {
      rand = random() % N ;
      sum += objs[randIndex]->A[rand] ;
    }
    PCQueuePush(bufQ, randIndex);
    countP++ ;
  }
}
```

Figure 1: Worker Thread and Prefetch Thread Code

3.2 Analysis

Let us start by considering how paging affects the system performance. Consider an access pattern in an application which repeatedly accesses some data and performs some computation on it.

N = the number of objects.	S = the size of each object in bytes.
M = the available memory.	t_c = the computation time.
t_p = page fault service time.	f_p = page fault frequency.
t = total completion time.	$t_0 = \frac{t}{N}$, effective time taken per object.

3.3 Large Computation Time

This section analyses different memory access patterns when the computation time per object is larger compared to page fault service time, i.e. $t_c > t_p$.

When all the required data is available in physical memory, access time is very small compared to computation time. Most of the time is spent in computation. The behavior is

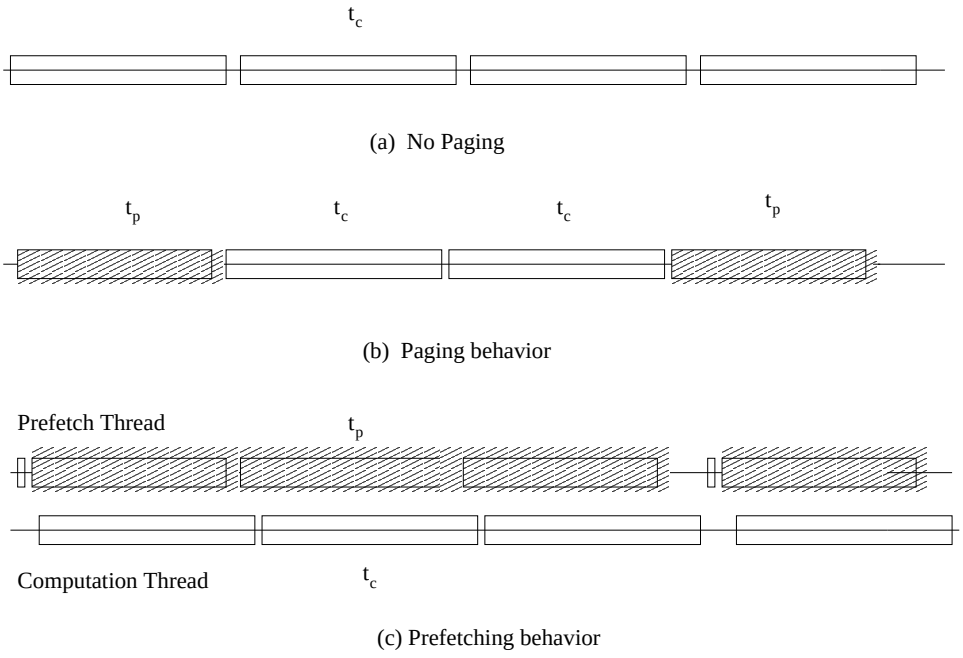

Figure 2: Timeline: Computation time higher than page fault servicing time

modeled in Figure 2 (a). Total time taken is given by the equation

$$t = N \cdot t_c$$
$$t_0 = \frac{t}{N} \; = \; t_c \tag{1}$$

As the amount of memory used in the application increases, not all the data will fit in memory and application might start paging. When a page fault occurs, time spent in access is high as compared to memory access with no page faults. This behavior degrades the overall performance of the application. Figure 2 (b) models the paging effect.

$$t = N \; (t_c + f_p \cdot t_p \;)$$
$$t_0 = \frac{t}{N} \; = \; t_c + f_p \cdot t_p \tag{2}$$

Let us calculate f_p now. When required memory is less than the available memory, i.e. $S \cdot N < M$, every access will be in memory, and there will not be any page faults.

$$f_p = 0, \; S \cdot N < M \tag{3}$$

As the problem size increases, we can define probability of a page not being in real memory as,

$$f_p = \frac{S \cdot N - M}{S \cdot N} \;=\; 1 - \frac{M}{S \cdot N} \quad , \; S \cdot N > M \tag{4}$$

Asymptotically, required memory reaches infinity and page fault frequency f_p approaches 1.

$$f_p \to 1, \; S \cdot N \to \infty \tag{5}$$

From equations 2, 3, 4 and 5, we have

$$\begin{aligned} t_0 &= t_c && , S \cdot N < M && (6)\\ &= t_c + (1 - \frac{M}{S \cdot N}) \cdot t_p && , S \cdot N > M && (7)\\ &\to t_c + t_p && , S \cdot N \to \infty && (8) \end{aligned}$$

From equations 6, 7 and 8, we conclude that time per object remains constant at t_c when the available memory is sufficient to fit the given problem, starts increasing with f_p for some time and then asymptotically approaches $t_c + t_p$ for infinitely large problem size. These equations are plotted as in Figure 3.

As seen in Figure 2 (b), computation thread needs to wait for the data to be brought in memory. The performance can be improved by overlapping page access with computation. To do so, a *prefetch thread* is introduced that will access any data before a computation is performed on it. At any point, when prefetch thread is blocked on a page fault, *computation thread* still continues to perform computation on earlier data prefetched. As seen in 2 (c), most of the computation is overlapped by page fault servicing time. Computation thread need not wait for next object access as in 2(b). We observe that computation time t_c dominates the total time taken. Thus, we conclude that

$$t_0 = t_c \tag{9}$$

From equations, 8 and 9, we conclude that asymptotically, t_0 will reach $t_c + t_p$ without prefetching, while with prefetching it remains constant at t_c. Figure 3 models the above analysis.

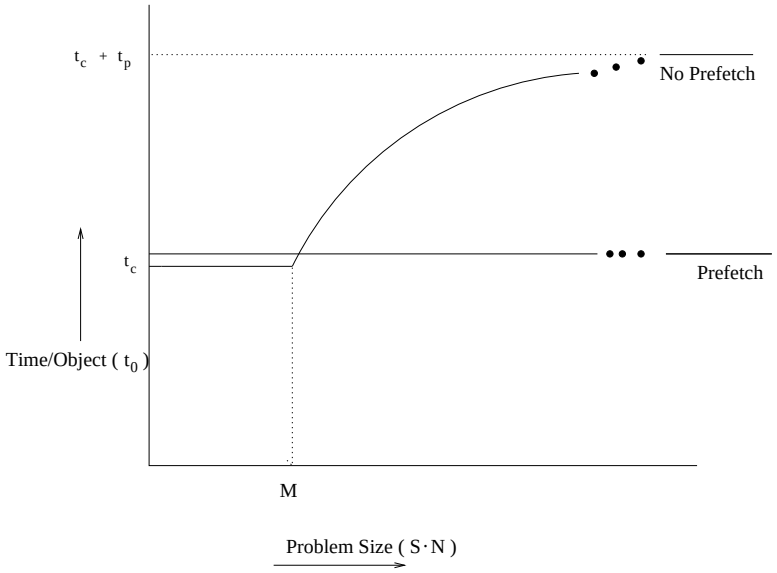

Figure 3: Large Computation Time: Time per Object Vs Problem Size

The break in the curve demonstrates the asymptotic behavior at infinite memory requirements. This analysis ignores second-order effects, such as

- the prefetch thread encountering multiple page faults in succession and thus making the computation thread wait although the average page fault rate isn't increased.
- a prefetched page getting replaced before it is accessed by the computation thread.

The slight difference between t_0 for prefetch thread and t_c is due to overheads of thread creation, thread context switches and redundant accesses by the prefetch thread.

3.4 Small Computation Time

The analysis in the previous section assumed that $t_c > t_p$. This section analyzes the scenario when the computation time is small compared to page fault service time. We observe that prefetching has advantages even when computation time is smaller than page fault service time.

Figure 4 models the behavior of the application with small computation time. The analysis for the case with no prefetching remains same as given by the equations 6, 7 and 8.

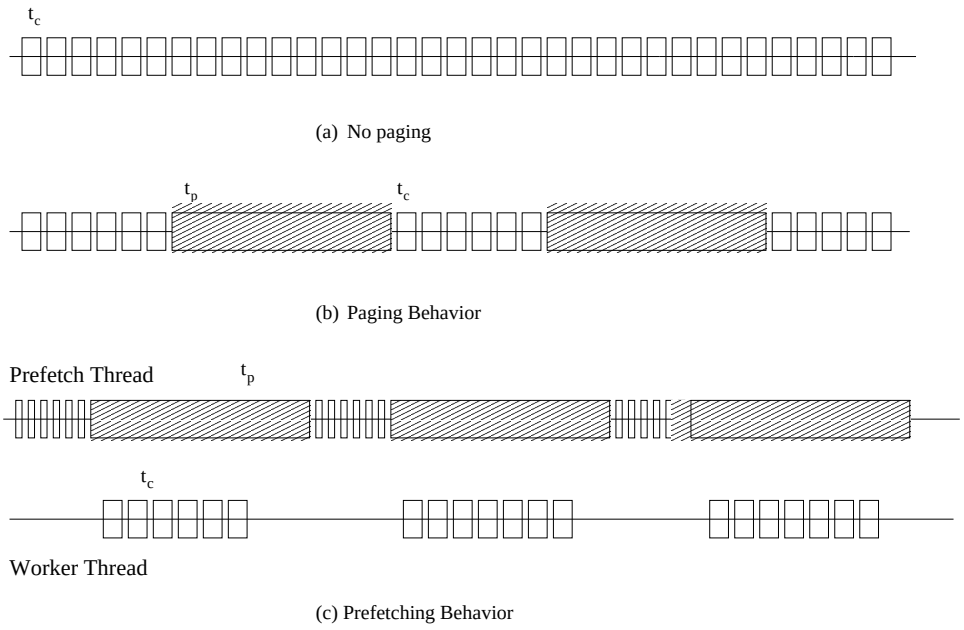

Figure 4: Timeline: Computation time smaller than page fault servicing time

When prefetching is introduced in such an application, an overlap is achieved between the paging time and the computation time as shown in Figure 4 (c). When $f_p \cdot t_p < t_c$, by the time the computation thread finishes executing K objects in time $K \cdot t_c$, the prefetch thread incurs $f_p \cdot K$ page faults, taking $K \cdot f_p \cdot t_p$ time units. Thus, assuming that the access time spent by the prefetch thread is negligible and so is the context switching overhead, the computation thread is never kept waiting by the prefetch thread. Therefore, we have,

$$t = N \cdot t_c \ , \ if \ f_p \cdot t_p < t_c$$
$$t_0 = t_c \ , \ if \ f_p \cdot t_p < t_c \tag{10}$$

This analysis also ignores second-order effects, as analysis in previous section.

If $f_p \cdot t_p > t_c$, on the other hand, the total time is decided by the page fault time. For a service of K accesses, the prefetch thread incurs $K \cdot f_p \cdot t_p$ time in servicing the page faults. The computation thread finishes its computations in time $K \cdot t_c$ which is less than $K \cdot f_p \cdot t_p$ during page fault servicing. Therefore,

$$t = N \cdot (f_p \cdot t_p), \ if \ f_p \cdot t_p > t_c$$

$$t_0 = (f_p \cdot t_p),\ if\ f_p \cdot t_p > t_c \tag{11}$$

The condition $f_p \cdot t_p < t_c$ can be rewritten as

$$\begin{aligned} f_p \cdot t_p &< t_c \\ (1 - \tfrac{M}{S \cdot N}) \cdot t_p &< t_c \\ i.e.\ S \cdot N &< M(\frac{t_p}{t_p - t_c}) \end{aligned} \tag{12}$$

According to our assumption, $t_c < t_p$. Thus, the quantity $(\frac{t_p}{t_p - t_c})$ is greater than 1. Thus, we observe that when prefetching is introduced in the program, t_0 remains at t_c for $(\frac{t_p}{t_p - t_c})$ times more than in program without prefetching.

Let us now consider the case when f_p approaches 1. According to equations 2 and 11,

$$t_0 \rightarrow t_c + t_p\ ,\ Without\ Prefetching \tag{13}$$

$$t_0 \rightarrow t_p\ ,\ With\ Prefetching \tag{14}$$

Thus, when computation time is high ($t_c \approx t_p$), prefetching makes the program twice as fast as the original program. The expected performance of the application is shown in Figure 5

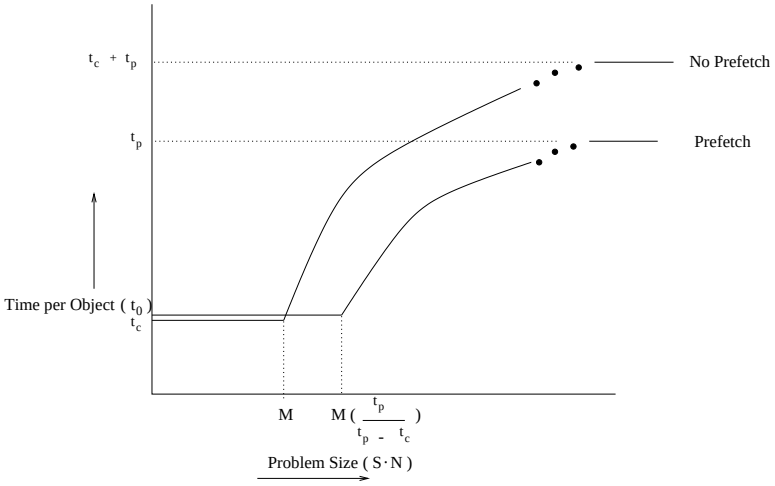

Figure 5: Small Computation Time: Time per Object Vs Problem Size

3.5 Asymptotic Behavior

Keeping N constant at very large value such that $f_p \rightarrow 1$, if we vary t_c, following graph can be plotted.

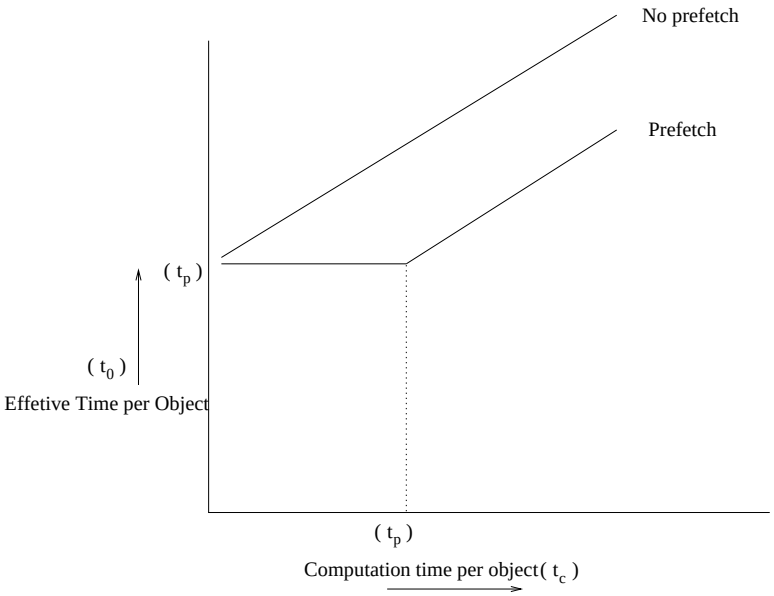

Figure 6: Asymptotic Analysis for different computation time per object

Without prefetching, effective time per object is given as $t_0 = t_c + t_p$, which is plotted as a straight line in Figure 6. When prefetching is introduced, t_0 depends on relative values of computation time per object & page fault service time. When $t_c < t_p$, effective time taken is dominated by the page fault service, $t_0 = t_p$, which is independent of t_c. When computation time is high compared to page fault service time, t_0 is dominated by the computation time, $t_0 = t_c$.

4 Performance Evaluation

The benchmark application we discussed in previous section is run on Linux and Solaris machines to study the paging behavior. The function *doWork* characterizes the work performed per object. It can be tuned to set time per object, t_c. Graphs are plotted between number of objects (N) and effective time taken per object, t_0. As we increase N, program starts paging

at some value of N. We study this behavior for different values of t_c, different leash sizes. We also study the asymptotic behavior of the program for different values of t_c. It takes several hours to complete all the data points in the graphs depending on the values of t_c and N. To perform more experiments in given time, we also wrote a simulator that will produce similar results in much shorter time.

4.1 Different Leash Size

Prefetch thread maintains a leash so that even when it is blocked at a page fault, computation thread can continue working. When the leash is 1, computation thread has to wait at every object until prefetch thread has enqueued it in the queue, showing no performance improvements. As the leash is increased, computation thread has more work to do when prefetch thread is waiting on a page fault. Thus, computation thread needs to wait for little or no time for work to be enqueued. Leash should not be selected so high that prefetch thread prefetches too much data which might get paged out before it is needed, thus causing even more page faults. Experimental results show that leash size of 10-30 show same performance benefits.

4.2 Large Computation Time

Figure 7 demonstrates the behavior of the program with computation time 20 ms. It shows that as N increases, effective time per object for *No Prefetch* program increases, but for *Prefetch* program it does not increase that much. The behavior is in accordance with equations, 8 and 9.

4.3 Small Computation Time

We discuss behavior of the benchmark application when computation time is small compared to page fault servicing time. Figure 8 and 9 show that for large values of N, difference between *No Prefetch* and *Prefetch* curve is approximately equal to t_c as in equation 13 and 14. Also, we observe that *With Prefetch* curve starts rising at a later point than the *No Prefetch* curve,

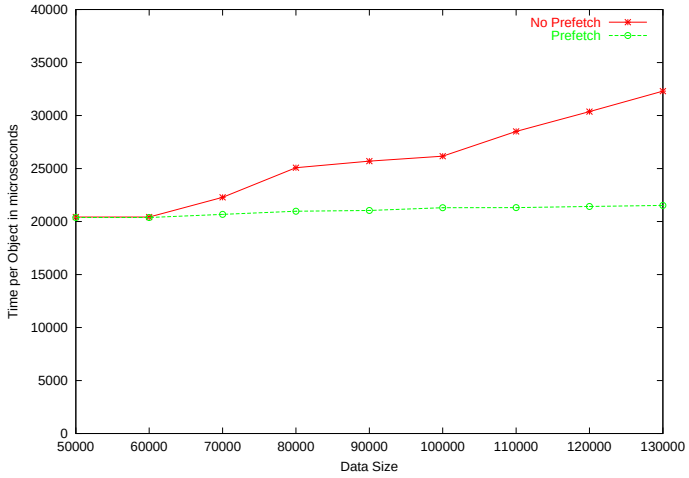

Figure 7: Runs on Linux with computation time of 20 ms

as expected in analysis done in equation 12.

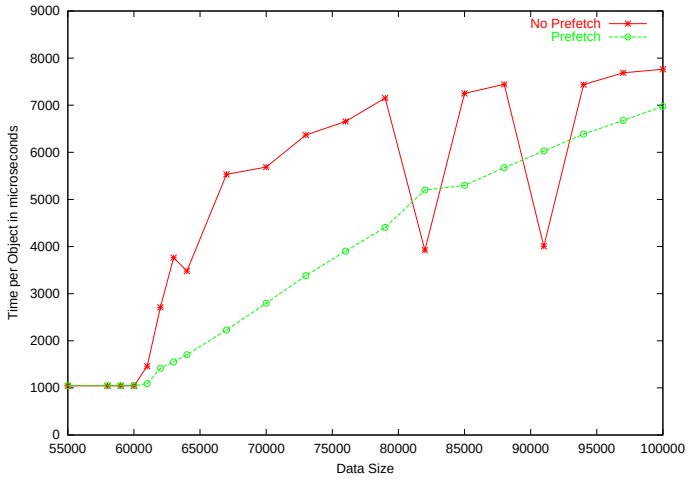

Figure 8: Runs on Linux with computation time of 1 ms

4.4 Runs on Solaris

Similar experiments were performed on Solaris to confirm the behavior of prefetching on Solaris. Experiments are performed with a fixed computation time of 1 ms while varying object data sizes. Graph is shown in Figure 10.

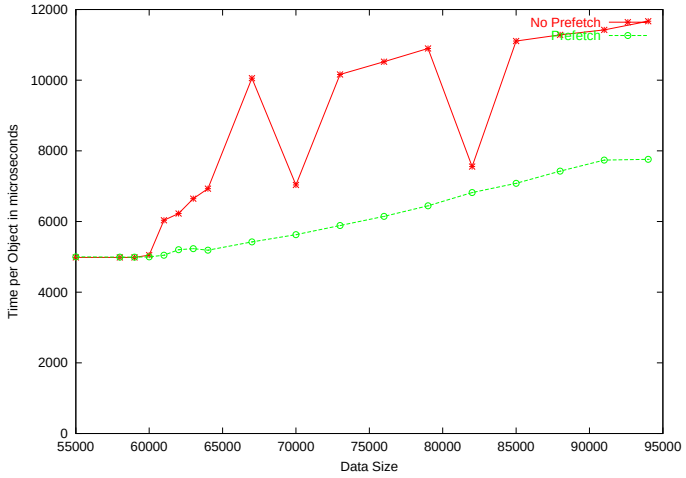

Figure 9: Runs on Linux with computation time of 5 ms

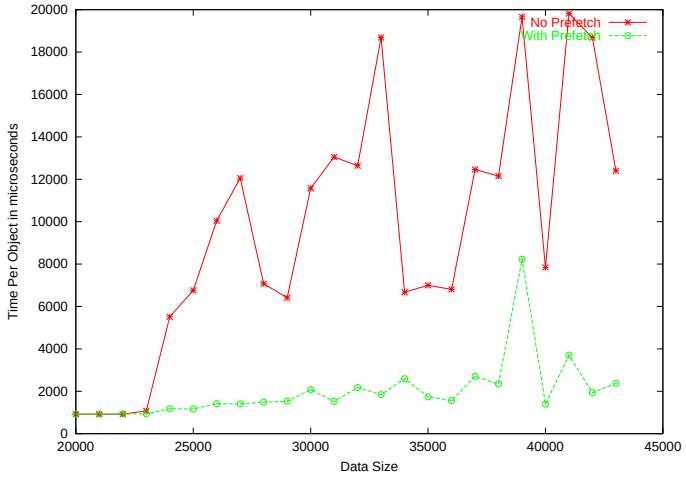

Figure 10: Runs on Solaris with computation time of 1 ms

4.5 Simulator

Benchmark application runs are limited by the length of time it takes to perform each run. It takes few days to perform a complete experiment to study a particular behavior. On the other hand, the equations we studied in section 4.1 ignore second order effects. Thus, we developed a a simulator which will take into account second order effects, but will not perform any actual work thus saving the execution time. The simulator does not perform any page accesses or any computation. It maintains two timer values, one for computation

thread and one for prefetch thread. It simulates the probability of the page faults depending on the data size. It also simulates the context switch between the two threads when queue is full or empty or when prefetch thread is taking a page fault. With the simulator, we can perform the experiment for much larger data size and computation time per object in much shorter time. The simulation data confirms the equations derived in section 3. Figure 11 plots the simulation data for computation time per object as 5ms.

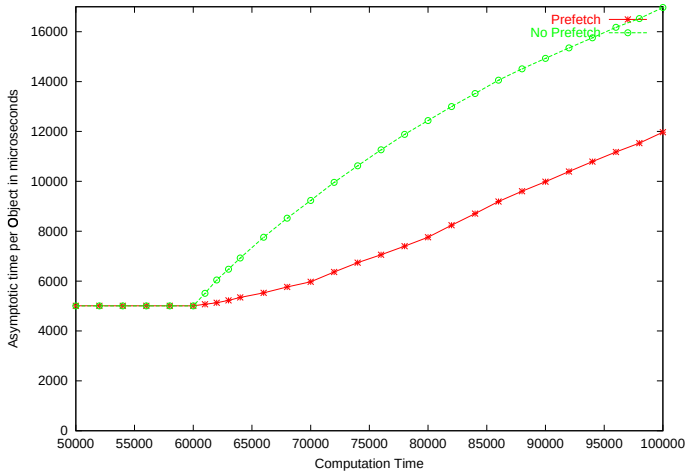

Figure 11: Simulator runs with computation time as 5 ms

Figure 12 shows the asymptotic behavior of programs when the object size is very high. The simulation environment assumes a large data size so that page fault rate is 95%. Computation time is varied from 1 ms to 40 ms. We can observe the change in prefetch curve as computation size increases beyond 20ms. The simulation graph is in accordance with the theoretical analysis in figure 6.

5 Summary and Future Work

For applications with large memory requirements, reducing or hiding paging costs is crucial. We proposed a multithreaded approach with a prefetch thread that accesses objects that are needed in near future, while computation thread is working on the objects available. data-driven object paradigms, such as CHARM++, facilitates this approach, since they

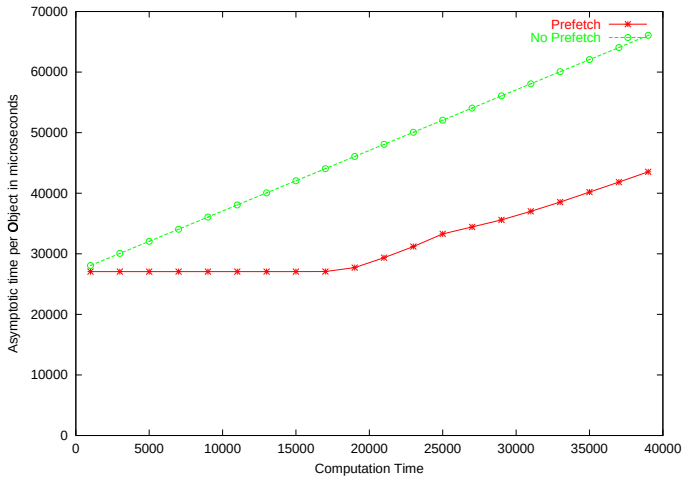

Figure 12: Simulation runs for asymptotic behavior with varying computation time

can "predict" the objects needed in near future. Benchmark application demonstrates the results of the prefetching concept and confirm that prefetching improves the performance of an application by overlapping paging and computation time. When computation time is higher than the page fault servicing time, computation time overlaps most of the paging costs and t_0 remains constant at t_c, while with no prefetching t_0 approaches $t_c + t_p$. For smaller computation times, prefetching performs better by t_c. Asymptotically, prefetching approaches t_p. Also, with prefetching, paging behavior shows effect at a larger data size hiding the effect of paging.

If more than one prefetch thread are introduced in the program, even when one prefetch thread is blocked on a page fault, other prefetch threads can keep prefetching objects producing more work for computation thread. Of course, the benefits will be limited by the ability of the paging system and disks to overlap multiple disk accesses. More experiments need to be performed for N prefetch thread approach. Also, interaction between paging behavior and context switch behavior of pthreads need to be studied more carefully, since we found several instances of inexplicable behavior in the course of this work.

To get explicit control of objects that are to be paged out and to avoid the overheads of thread creation and context switching between threads, we plan to experiment with an

approach using asynchronous I/O mechanisms. Objects are serialized and stored on disk. To prefetch an object, it is asynchronously read, deserialized and brought in memory. Asynchronous operations can be overlapped with the computation. This approach of prefetching needs more careful design and feasibility study. To compete well with the other explicit out-of-core approaches [4], [5], we plan to experiment with schedulers that are aware of which objects are in memory and reorder method invocation to minimize paging.

References

[1] Lewis Bil and Berg Daniel. *Multithreaded programming with PTHREADS.* Prentice Hall, 1998.

[2] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[3] L.V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *International Parallel Processing Symposium 1996 (to appear)*, 1996.

[4] Todd C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on computer Systems*, 16(1):55–92, February 1998.

[5] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. 30(8):1–10, August 1995.

[6] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceddings of the 1995 International Conference on Parallel Processing*, pages 29–33, 1995.