Profile-guided Proactive Garbage Collection for Locality Optimization

Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{wenkec, sanjaybh, trishulc}@microsoft.com

Xiaofeng Gao, Weihaw Chuang
Dept. of Computer Science and Engineering
University of California at San Diego
La Jolla, CA 92093
{xgao,wchuang}@cs.ucsd.edu

ABSTRACT

Many applications written in garbage collected languages have large dynamic working sets and poor data locality. We present a new system for continuously improving program data locality at run time with low overhead. Our system proactively reorganizes the heap by leveraging the garbage collector and uses profile information collected through a low-overhead mechanism to guide the reorganization at run time. The key contributions include making a case that garbage collection should be viewed as a proactive technique for improving data locality by triggering garbage collection for locality optimization independently of normal garbage collection for space, combining page and cache locality optimization in the same system, and demonstrating that sampling provides sufficiently detailed data access information to guide both page and cache locality optimization with low runtime overhead. We present experimental results obtained by modifying a commercial, state-of-the-art garbage collector to support our claims. Independently triggering garbage collection for locality optimization significantly improved optimizations benefits. Combining page and cache locality optimizations in the same system provided larger average execution time improvements (17%) than either alone (page 8%, cache 7%). Finally, using sampling limited profiling overhead to less than 3%, on average.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *code generation, memory management (garbage collectors), optimization, run-time environments*

General Terms

Measurement, Performance, Experimentation.

Keywords

data locality, garbage collectors, cache optimization, page optimization, memory optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'06, June 11–14, 2006, Ottawa, Ontario, Canada.
Copyright 2006 ACM 1-59593-320-4/06/0006…$5.00.

1. INTRODUCTION

Many programs, especially those that manipulate pointer data structures, are memory performance limited due to the growing disparity between processor speeds and memory access times [10]. Large, multi-level caches help hide some of the memory access latency and translation look-aside buffers (TLBs) mitigate the page translation costs. Unfortunately, caches and TLBs are expensive and unlikely to grow at the same rate as application workloads, especially on-chip caches. Perhaps most importantly, these hardware mechanisms are often limited by poor program data layout, which rarely takes full advantage of the multi-word data transfer granularity that cache lines (64 – 128 bytes) and pages (4 – 8K bytes) provide.

Two useful layout metrics are page and cache line density, which indicate how well program elements are laid out and packed together. Unfortunately, as Figure 1 indicates[1], many programs have poor page density. In addition, programs also have poor cache line utilization, averaging around 30% [24]. This provides an opportunity to make more efficient use of caches and TLBs by packing contemporaneously accessed data elements together and reducing a program's page and cache footprint.

Newer mainstream languages such as C# and Java support automatic memory management, which is implemented with a garbage collector that, when necessary, automatically examines the program heap and recycles space occupied by dead data for use in subsequent allocations. To avoid heap fragmentation, many garbage collectors either copy all live objects or compact the heap by moving some live objects into the space freed up by dead data. Garbage collectors maintain fairly elaborate infrastructure to accomplish this task. Prior research has leveraged this infrastructure to combine this object movement with intelligent placement to optimize either program page or cache locality [6,7,8,13,16,20,22].

This paper describes a locality optimizing system that leverages the garbage collector with three key differences from earlier work. First, it uses sampled profiles of data accesses to drive both page and cache locality optimizations. Second, it proactively triggers

[1] These numbers were obtained by running these applications using a dynamic translator and logging all memory reads and writes. References to stack pages were filtered out.

garbage collection for locality optimization rather than passively performing locality optimization only when garbage collection is invoked due to memory space constraints. In addition, it implements an automatic online throttling scheme that limits performance degradation for applications that do not benefit from locality optimization. Finally, it combines page and cache locality optimization in the same system.

Placing contemporaneously accessed objects together at the page and cache level requires accurate data access information. While static analysis techniques continue to improve, they are still unable to provide sufficiently detailed data access information for this purpose, especially for large programs that manipulate pointer data structures. Hence we monitor data accesses at run time and use this information to guide data placement. Unfortunately, the runtime overhead can sometimes be too high. To address this, we use bursty tracing [2][5], which is a form of sampling that captures fine-grain temporal data access information, to reduce our profiling overhead to less than 3% on average, yet produce sufficiently detailed data access profiles to guide locality optimization. While previous research on using garbage collection to improve cache locality in the Cecil system found profiling without sampling had sufficiently low-overhead [6], our experience with a commercial system indicates otherwise. While several differences between the two systems makes comparison difficult (different languages, SPARC (32 registers) Vs. x86 (only 8 registers)), a possible reason for this is our implementation platform is a highly-tuned commercial system that is less tolerant of instrumentation overhead than the Cecil research prototype.

Garbage collection is typically triggered in response to an allocation request when the amount of free space falls below a certain threshold. This can often coincide with a program phase boundary especially if the new phase starts by allocating many new objects. If locality optimization (LO) is passively combined with garbage collection (GC), it would incorrectly place objects that were contemporaneously accessed in the previous phase together, possibly reducing program locality for the current phase. To avoid this problem and to be able to continuously reorganize the data layout in response to changes in data access patterns caused by program phase behavior, we decouple LO from GC and enable triggering LO independently of GC. We use metrics such as cache and TLB miss rates obtained from hardware performance counters as well as allocation behavior to trigger LO. In addition, we have implemented an automatic online throttling scheme that limits performance degradation for applications that do not benefit from LO. Our results indicate that actively triggering LO independently of GC provides significant performance benefits.

Application	*Pages touched per interval*	*Average Page Density*
Web page renderer 1	600	7.7%
Web page renderer 2	588	6.5%
XamlParserTest	602	6.0%
Sat Solver	1703	28.0%
Compress	102	28.0%

(a) C# applications

Application	*Pages touched per interval*	*Average Page Density*
Multimedia App 1	519	7.1%
Multimedia App 2	264	35.4%
Desktop App 1	368	7.3%
Desktop App 2	367	9.5%
Desktop App 3	315	13.2%
Internet App	478	11.2%

(b) C/C++ applications

Figure 1. Application page density (combination of Microsoft and non-Microsoft applications). The density of a page = numbers of unique bytes read/written on a page per interval / size of page. An interval is chosen to be 10^6 references.

Our LO system addresses both page and cache locality. To optimize page locality, object accesses are recorded by setting a bit in the object header. In addition, to optimize cache locality we record the object address in a fixed size circular object access buffer. During LO, live objects that have their access bit set are copied and placed contiguously according to a hierarchical decomposition order [20] (called page locality placement herein), which improves page locality, and for some of the applications we studied, also improves cache locality. To specifically improve cache locality, we use a similar scheme to that described in [6], which uses the object address buffer information to place contemporaneously accessed objects together (called cache locality placement herein). Since the object address buffer is of fixed size, it only contains a subset of objects that were accessed since the last LO. Our LO combines page and cache locality optimization by first performing the cache locality placement for objects in the object address buffer followed by the page locality placement for objects whose access bit was set but did not appear in the object address buffer.

We implemented our LO system in the Common Language Runtime v2.0 (CLR) of Microsoft's .Net Framework. Our choice of implementation platform and benchmarks was driven by our goal of transferring this technology to Microsoft's commercial platform. All of the applications we used for benchmarking are written in C#; however, the results are applicable to any language that targets MSIL (Microsoft Intermediate Language) binaries.

The main contributions of the paper are:

1) Decoupling LO from GC and demonstrating benefits of triggering these independently. In addition, we implemented an automatic online throttling scheme that limits performance degradation for applications that do not benefit from LO. We show that this technique significantly improves average optimization benefits due to improved mutator locality.

2) Combining cache and page locality optimizations in the same system and demonstrating performance gains. Earlier research either performed cache or page optimization, but not both. For the C# applications we studied, combining page and cache locality optimization in the same system provides larger average

improvements (17%) than either alone (page 8%, cache 7%).

3) Demonstrating that sampling techniques can be used to collect sufficiently detailed data access information to guide cache and page locality optimizations with low overhead (less than 3% on average).

4) Implementing and evaluating the system in a commercial managed runtime system. We implemented this in Version 2.0 of the Common Language Runtime GC, which ships with Microsoft's .NET Framework.

We believe that automatic locality optimization techniques such as these are necessary for modern languages, such as C# and Java, to approach and perhaps even surpass the performance of C/C++ programs.

The rest of the paper is organized as follows: Section 2 briefly discusses related work. Section 3 describes the design and a few implementation details of our technique. Section 4 contains experimental evaluation of our approach using several C# applications. Section 5 summarizes the main results and directions for future work.

2. RELATED WORK

The idea that garbage collection could be used to improve a program's locality was proposed as early as in 1980 by White [18]. Zorn speculated on the possibility of doing garbage collection purely for program performance in [9]. Several researchers have proposed ingenious traversal algorithms for copying garbage collectors to improve the locality of references in the collected heap [16] [20]. Wilson [20] proposed hierarchical decomposition to group all structurally related objects together to improve locality. We similarly group only recently accessed objects for page locality optimization. Shuf et al [21] proposed a new allocation scheme to improve locality by placing objects based on the notion of prolific (frequently instantiated) types. These approaches are based on static or offline profile information, and the virtual machine or runtime involved is not active in observing the dynamic data access sequence and determining how objects should be placed in the heap. Thus the layout derived may not reflect actual data access patterns, and moreover, it is not possible to detect phase changes and react accordingly.

Huang et al. [22] use a technique called online object reordering (OOR) that uses sampling to identify hot methods, and from these hot fields and their types. At garbage collection time, the GC copies referents of hot fields together with their parent, guided by the hot types. Their work is complementary to ours. Since their work was done on a virtual machine with function profiling built in and they sample hot methods rather than contemporaneous data accesses, their overhead is low. However, they are limited to coarse-grained profiling information about data accesses and in the type of placement optimizations they can perform. In addition, they perform their placement optimization during normal GC traversal. We trigger locality optimization independent of normal GC and show that this benefits performance. In addition, we optimize separately for both cache and page locality in the same system.

Courts described a dynamic approach in [8]. In his implementation, memory is divided into regions based on generation and activity. The GC copies inactive objects out of active space based on object accesses in the training period before each garbage collection. However, his implementation relied on the *transporter*, a micro-coded system service, to bring the objects from inactive space to active space when they are first accessed. In his system, garbage collection is just a passive activity that is triggered based on memory pressure.

Chilimbi and Larus proposed a scheme that uses online profiling to construct an object affinity graph from observed accesses, and then uses the GC to rearrange the objects for better cache locality [6][7]. We use their scheme to optimize cache locality but combine it with page locality optimization. We incur less runtime overhead since we sample data accesses. Finally, we trigger LO independently of GC while they perform cache optimization only when GC is invoked due to memory pressure.

Adl-Tabatabai et al [1] insert prefetch instructions in JIT compiled code and use GC's placement ability to maintain the distances of objects so that the prefetch is effective. They also use hardware performance monitoring counters to collect cache miss profiles to determine the prefetch sites. Our work is complementary as we instrument the application to detect hot objects and re-arrange these hot objects to improve their reference locality instead of trying to prefetch objects that will be accessed in the future. Our approach increases the spatial locality of frequently accessed objects and consequently increases the effectiveness of hardware prefetching.

Hertz et al [11] describe a technique that avoids paging by integrating the garbage collector with the memory manager so the GC can make informed decisions about evicting pages. That approach is orthogonal to ours since they do not address the issue of intra-page layout.

Our low overhead design relies on having cheap read barriers proposed by others [3], but the implementation we describe is quite different: the CLR implementation does not use null checks but implicit traps to detect null dereferences, the object headers do not have an extra to-pointer, and the heap implementation is quite different (for object headers that cannot be moved and the objects that are moved the to-space). Our compiler optimizations to reduce the number of read barriers are similar.

3. DESIGN AND IMPLEMENTATION

Our approach relies on a generational garbage collector that moves objects, and our prototype ties tightly with the CLR and MSIL (Microsoft Intermediate Language), but is otherwise agnostic of the implementation details of a virtual machine. In this section, we first provide a high-level overview of the approach, discussing various options and the rationale for our adopted design, and then describe the engineering optimizations necessary to provide a practical implementation.

Figure 2 provides an architectural overview of our design. A JIT compiler takes an intermediate language representation (MSIL in our case) and compiles it into machine code for a particular architecture. We modify the JIT compiler to insert lightweight instrumentation in the compiled code. The CLR provides a tool called Ngen [25] that can compile modules ahead of time to provide more optimized code and improve startup performance. Our modifications to the compiler work seamlessly with Ngen, so there is no extra startup penalty for using our approach. The instrumented code marks objects that have been recently accessed

and records their address in a fixed size circular object access buffer. We insert monitoring code in the CLR to gather certain metrics while the application is running and use the collected data and heuristics to trigger GC-for-locality, or Locality Optimization (LO). During LO, we use the Chilimbi-Larus scheme for combining cache locality optimization with GC. In addition, we perform page locality optimization for the objects that are not moved during cache locality optimization. To do this, we identify objects that have been marked as recently accessed (hot) and co-locate them according to their inherent structural relationship onto pages separate from the rest of the heap. LO is triggered independently of a normal GC, which occurs when the allocation budget is about to be exceeded. To limit performance degradation for applications with poor algorithmic locality, we throttle LO when indicated.

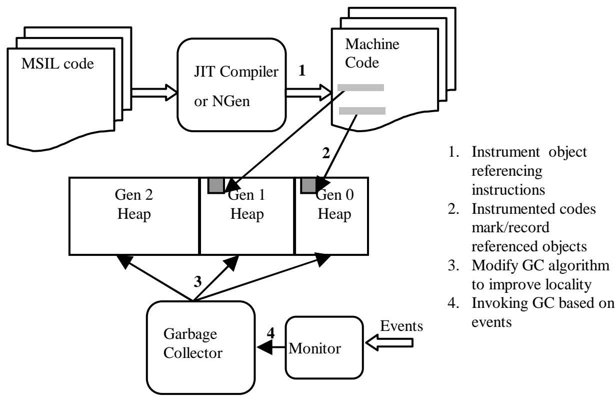

Figure 2. Overview of architectural modifications to the CLR.

We first provide a brief review of the scheme used to optimize cache locality [6]. Then, we discuss some of the key design issues, design options that we considered, and our rationale for adopting the current design.

3.1 Background: Cache Locality Optimization

We record base object addresses in a fixed size circular buffer. During Locality Optimization (LO), these objects are processed to construct an object affinity graph. The object affinity graph includes edges between objects that are contemporaneously accessed, where objects accessed up to 3 unique accesses apart are considered contemporaneous. Graph edges are weighted to indicate how often the objects have been accessed together. The garbage collector performs a weighted DFS traversal of this object affinity graph and marks objects for copying to a temporary buffer in the order they are visited. Since these objects may include some garbage (though prior work indicates that the amount is miniscule [6]), they are removed when the objects are physically copied in a later phase of the same garbage collection. The object affinity graph is not maintained across LOs but is created from scratch at each LO. Further details are in [6]. Our page locality optimization, which is described in the next few sections, copies objects to the end of this temporary buffer used by cache optimization.

3.2 Low-overhead Data Access Profiling

Since the instrumentation for cache locality is virtually identical to that described in [6], we focus on the instrumentation needed for page locality optimization and discuss our use of sampling to reduce profiling overhead.

3.2.1 Instrumentation for Page Locality Optimization

For page locality, we do not need to determine precise temporal affinity between data elements. We record objects that are frequently accessed during a time interval; these objects are considered hot and we group them during optimization according to their inherent structural relationship into a set of pages in a separate section of the heap. A counter is used to decide which objects are hot.

We use the JIT compiler to insert read barriers for certain critical instructions that access heap data. The read barrier code consists of a single call instruction to a helper routine which updates the counter if necessary. Write barriers are automatically generated by the compiler to support the generational GC, and we simply modify those to insert a conditional update of the counter.

The key engineering decisions we had to make were:

- Implementation of the counter
- Implementation of the read barrier
- Optimizing the instrumentation

We considered two ways to implement the object reference counter: embed it in the object or implement it as a separate table. Our current implementation uses a 1-bit counter that is embedded in the object. The CLR already has a four-byte object header for each object that is used for various purposes (e.g. to implement a lightweight lock or to store a hash code for the object). We modified this layout to steal one bit for our purposes. Although this reduces the number of bits available for other purposes, we feel that this is a good performance tradeoff. The main impact of stealing the bit is reducing the number of objects whose hash can be stored in the object header (from 27 to 26 bits) and reducing the number of concurrent threads that can be supported with lightweight locks. When the bits overflow they are repurposed to index into a table that points to larger object headers.

The read barrier code is shown in Figure 3.

```
   test     dword ptr[rg-4], OBJECT_ACCESSED_BIT
   jnz      Bit_set
   lock or  dword ptr[rg-4], OBJECT_ACCESSED_BIT; atomic update
Bit_set :
   ret
```

Figure 3. Profiling code used to mark accessed objects for page locality optimization. rg is the register that holds the object address. The object header is at offset -4 from the start of the object. OBJECT_ACCESSED_BIT is a bit mask used to set a single bit in the object header.

We use an interlocked operation to set the bit since the object header could be concurrently modified by other threads on an SMP machine. The interlocked operation is expensive on x86

architectures (20-30 clock cycles). In addition, it dirties a cache line during a read operation that could hurt scalability of applications on multi-processors. Therefore we implement a conditional read barrier instead of an unconditional one even though it bloats the read barrier code quite a bit. To minimize code bloat we do not inline the read barrier but implement it as a helper routine (one for each register). [2]

Algorithms used for optimizing access barriers can be directly applied here to further reduce the number of read barriers, and hence, the amount of code bloat. The read barrier used here is different from typical access (read or write) barriers in that we do not need to insert a call to it at every access point, because it does not affect the correctness of the generated code. This allows us to perform more aggressive optimizations. In our prototype implementation, we use common sub-expression elimination (CSE) to optimize away redundant calls to the read barrier routines. Furthermore, since occurrences of exceptions are rare, no profiling calls are inserted into exception handling code. Similarly, we also ignore constructors that are not inlined.

One policy decision we had to make was when to reset the counter. Because of our decision to embed the counter in the object, we cannot clear the bit without a scan over the whole heap, which is expensive. The only natural opportunity for clearing the counter is when we do a GC. We experimented with a few different schemes and found that the simple strategy of clearing the counter every time we encounter a hot object during a GC (no matter whether for locality or for space) works well for objects in lower generations (generation 0 and 1). For higher generations (generation 2 in CLR) this does not work as well because those generations are collected infrequently and the reference bit gets stale over time. In our prototype, since the collector does need to check generation 2 objects that contain cross-generation pointers, we use this opportunity to clear the counters embedded in them, which alleviates the problem to some extent. An alternative would be to store the counters in a card table, which would make clearing the counters relatively cheap.

3.2.2 Sampling Data Accesses

The page locality instrumentation model described above has low overhead and is enough to speed up some benchmarks that we describe later. But there are several scenarios where dynamic heap reorganization does not help improve the performance of an application, e.g., if the dataset is small enough to fit in the available memory or the algorithm has poor locality. For such applications the cost of the read barrier can be very high (in some cases degrading the application by as much as 40%). In addition, the overhead of the instrumentation needed by our cache locality optimization is high.

To further reduce the instrumentation overhead, we use a simplified version of bursty tracing [2][5]: if a method is instrumented with a read barrier, we generate a second copy of the method without the read barrier. The prolog of each copy of the same method is extended to perform a check and control transfer to either the instrumented or the non-instrumented version of the method. Back edges are not modified in our simplified implementation. Surprisingly, this simplification does not reduce the effectiveness of this approach on the benchmarks we examined (except for some synthetic ones that have long-running loops); the reason is that modern software practices and object-oriented programming languages usually result in many more smaller functions than larger ones, where deeply nested loops are rare. As a further optimization, the two copies are placed in separate code heaps.

There are two parameters to control the sampling: how long each burst should last and how often sampling should be triggered. By tuning these two parameters, we can obtain useful profile information at a reasonably low profiling overhead. Our experiments show that with this bursty tracing scheme, we can limit pure profiling overhead to less than 5% - the cost of doing the check in the prolog.

3.3 Combining Page and Cache Locality Optimization

The CLR GC implements a variation of generational mark-compact garbage collection. It divides the small object heap into three generations, and moves live objects into older generations in their allocation order when triggered [15]. We modified the implementation so that GC can be independently triggered either for space or for locality optimization. However, when GC is triggered, unless the policy says it is for space only, it will attempt to do both at the same time, with one exception: when it is triggered to collect only Generation 0 objects, locality optimization will not be applied. The rationale for not doing heap reorganization for locality optimization during a generation 0 collection for space is that most of those generation 0 objects, being recently allocated, are already hot and in the cache or working set, and are unlikely to benefit much from locality improvements. In addition, many of these objects are likely to die shortly. During a GC for locality we identify all objects that a) have their address entered in the circular object access buffer (cache locality optimization) b) were marked as hot since the previous locality collection (page locality optimization) and c) belong to a generation not older than the generation being collected. Only these objects are candidates for locality optimization.

After all the candidate objects having been identified, the locality optimization needs to decide how they should be laid out and where to put the hot objects on the GC heap. To simplify our implementation, we do the layout using two copying phases for the hot objects. First, we perform cache optimization by copying contemporaneously accessed objects to a temporary buffer. Next, we perform page optimization by copying and appending heap objects marked as hot into this same buffer according to a hierarchical decomposition order based on their inherent structural relationship [20]. This can also yield some cache locality benefits along with page locality. The original locations are marked free and reclaimed by the collector. The well-rearranged aggregation of hot objects is then placed back at the younger end of the heap (either Generation 1 or Generation 0).

We considered other schemes that could avoid the double copying (e.g. by reserving a designated section of the heap), but discarded them because of several complications (e.g. CLR supports the notion of pinned objects) in the implementation. We also considered other layout schemes that did not mix objects from different generations, but finally decided to use our current scheme for the following reasons: (1) we are guaranteed to have

[2] The code bloat is < 3% for the applications we studied.

enough space at the younger end to accommodate all the hot objects; (2) we don't want to promote objects prematurely, because it is more expensive to collect an older generation than a younger one; and (3) some longer-lived objects tend to die right after being reused, and demoting will accelerate the reclamation of the space occupied by these objects. In general blindly demoting many objects is not good, but we do this selectively for hot objects (which comprise a small fraction of the heap).

We also make sure that it does not create too many cross-generational pointers because that will make it more expensive to collect younger generations, which usually happens more frequently. We compute the number of cross-generation pointers that will be created before finalizing our optimization and back-off if this exceeds a predetermined threshold. (In our prototype we use 6,000 which worked well.)

3.4 Triggering LO independently of GC

Garbage collection can often coincide with a program phase boundary especially if the new phase starts by allocating lots of new objects. Passively combining locality optimization (LO) with garbage collection (GC), as is traditionally done, would incorrectly place objects that were contemporaneously accessed in the previous phase together, possibly reducing program locality for the current phase. To avoid this problem and to be able to continuously reorganize the data layout in response to changes in data access patterns caused by program phase behavior, we decouple LO from GC and enable triggering LO independently of GC.

One challenging aspect is to automatically determine conditions for triggering GC for locality (LO) as well as conditions for determining when to back off, e.g. when the optimization is not working as well as anticipated. We have implemented and experimented with several different strategies for both. For triggering LO we tried the following options:

a) Use hardware performance counters, e.g., do LO when the rate of DTLB and L2 cache misses increases by certain amount.[3]

b) Use rates of object allocation, e.g., do LO when there is a significant drop in allocation rate as it likely indicates that the application is done "setting up" the new phase.

We also tried combinations of these heuristics. One drawback with using hardware performance counters is that they are not virtualized to a process on many current generation machines (e.g. Intel's x86 family of machines) and so the numbers could be skewed by other applications running on the system. We found that the rate of object allocation is a reliable measure for triggering LOs. We experimented with several heuristics and the one listed below provided the best experimental results across our application test suite (see Section 4.3.2 for details). We use allocation rate as the primary trigger for heap re-organization for locality and additionally consult the DTLB and/or L2 cache miss rate when the allocation rate remains relatively stable: if the allocation rate drops by more than 12.5%, do a Generation 1 LO collection, if it drops by more than 50% do a Generation 2 LO collection; otherwise, if either the DTLB or L2 cache miss rate (computed from data read from the hardware performance counters) increases by 6.25% / 25% do a Generation 1 / Generation 2 LO collection; otherwise, LO will be done along with GCs triggered for space.

We currently use a simple scheme for backing off and throttling LO when locality optimization appears ineffective. If neither DTLB nor L2 cache miss rates have improved by 5% over their historical value immediately following a LO, for two successive LOs, we disable LO for the next few GCs. The number of GCs for which LO is disabled starts at 2 and is exponentially increased until LO improves DTLB or L2 cache misses by at least 5%, at which time it is reset to 2. This simple scheme worked well in practice as discussed in Section 4.

4. EVALUATION

This section presents and analyzes the results from experiments done with our prototype implementation.

4.1 Experimental Platform

As mentioned earlier, our prototype is based upon version 2.0 of the commercial CLR implementation on the Windows XP operating system. We did not modify the CLR GC's heap allocation budgets, its policy for determining when to grow the heap, and the algorithms used to determine the sizes of the various generations since those policies have been highly tuned and are very complicated to modify [15]. In addition, our goal was to investigate the impact of LO on a well-tuned GC. We performed experiments on several machines with different memory, cache size, and processor speed configurations. Unsurprisingly, we found that our locality optimization works much better on machines with smaller L2 cache and memory. However, we believe that for real-world scenarios where performance matters, machines will be configured to have adequate memory and large L2 caches. Hence, results reported below were obtained on a machine with the following configuration:

CPU: Pentium 4, 2.8 GHz
DTLB: 64 entries
L2 cache: 1MB, 8-way, 128-byte cache line
RAM: 1GB

4.2 Benchmarks

Due to the lack of widely available benchmarks for the .NET framework, we obtained six large applications written in C# that are used internally at Microsoft[4]. These C# applications, which we obtained from our colleagues, are briefly described in Table 1. The number in parenthesis in the Original Time column is the percentage of execution time spent in garbage collection.

4.3 Performance results and analysis

We performed three sets of experiments. First, we measured the profiling overhead of gathering data access information for our optimizations. Next, we evaluated the benefits of triggering LO independently of GC. Finally, we measured the benefits of our page and cache locality optimization.

[3] We wrote a kernel-mode driver that allowed the hardware performance counters to be read by the application on demand.

[4] Two of these applications have been made externally available at http://research.microsoft.com/~zorn/benchmarks/

Table 1: C# Benchmark Descriptions.

Name	Description	Input	Orig. Time in secs (% time in GC)
Xaml-Parser-Test	Reads from an XAML (extensible application markup language, based on XML) file three times to measure the performance of different components of the parser.	11,000-level deeply nested node	100.0 (0.2%)
SAT-Solver	SAT solver in C# ported from a C++ implementation.	Problem instance with 24,640 3,250-variable CNFs	138.0 (0.4%)
Max	Analyzes and builds a dependency relationship among a set of modules specified in an XML file.	3338- module input	97.7 (26%)
GenIBC	Computes optimal object layout from profile data.	A 55 MB xml file	6.5 (27%)
Fugue	.NET protocol checker that checks managed API usage rules	A 2MB compiled .NET assembly	79.9 (10.3 %)
Lcsc	A C# front-end that generates MSIL code	A 125K LOC C# file	42.6 (9.1%)

4.3.1 Profiling Overhead

As mentioned in Section 3, we applied several static optimizations, such as CSE, to reduce the number of instrumented data access sites. These cut down the number of data accesses that require instrumentation by a factor of two on average. In addition, the worst case code bloat due to instrumentation was less than 3%.

We investigated several sampling rates to pick one that provides a good tradeoff between overhead and profile accuracy. Figure 3 illustrates the overhead results. Since we use bursty tracing, we can vary both the overall sampling rate and the burst length. We evaluated a wide range of values for both parameters but only report results for sampling rates of 0.1%, 0.05%, 0.01% and burst lengths of 100 and 1000 units (where each unit is 64K clock cycles) as these significantly outperformed the rest in terms of overall optimization benefits. Higher sampling rates introduced larger overheads that our optimizations were often not able to overcome and lower rates degraded the performance impact of our optimizations. Similarly, larger burst lengths increased overhead without improving optimization benefit and shorter bursts negatively affected cache locality optimization. All subsequent experiments use a sampling rate of 0.05% with a burst length of 100. This sampling rate not only resulted in profiling overheads that were less than 3% on average, but produced sufficiently accurate profiles to drive the optimizations as reported in the following sections. While prior research has indicated that bursty tracing provides low-overhead with good profile accuracy for Java and C/C++ applications [2, 5], we wanted to ensure that our variant of bursty tracing performed as well for C# applications.

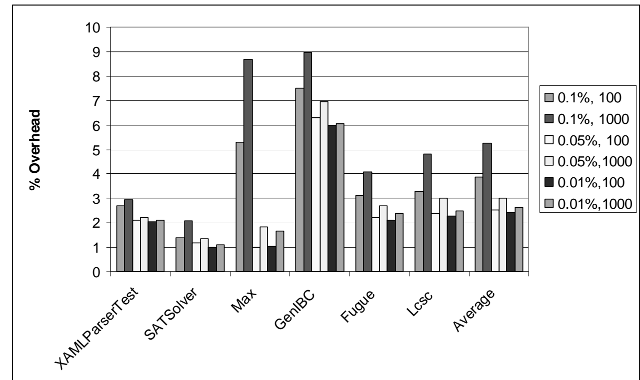

Figure 3: Evaluating different sampling rates and burst lengths.

Figure 4 indicates the overall impact of these techniques on profiling overhead. Always On (Page) represents the overhead of profiling for page locality with static optimization but no sampling and Always On (Page + Cache) represents the profiling overhead of gathering data for page and cache locality optimization. As the data indicates, sampling is essential for reducing profiling overhead in our system. For our C# applications, profiling overhead is less than 3% on average.

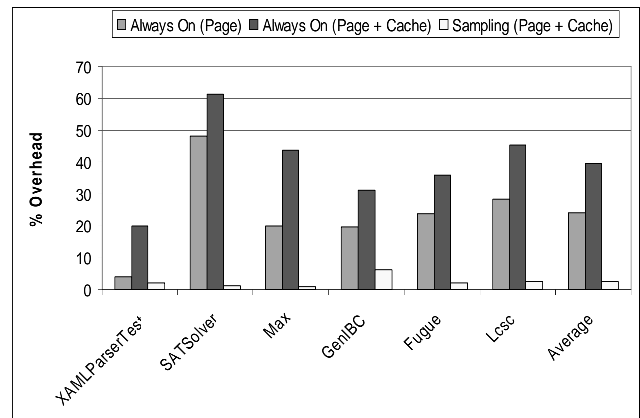

Figure 4: Profiling overhead for C# applications.

4.3.2 Triggering LO independently of GC

The next set of experiments indicates the benefit of separating locality optimization (LO) from GC and triggering them independently (Pro-active LO). Triggering LO independently of GC increases the total number of GCs performed since it does not passively wait for memory budget pressure to invoke GC. In addition, it changes when GC is performed.

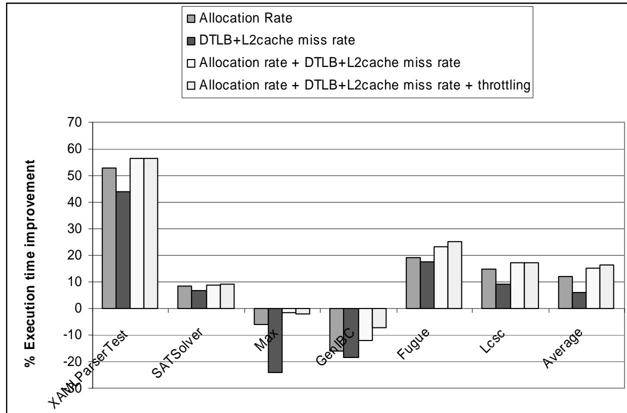

Figure 5: Comparison of different LO triggering policies.

We evaluated several different schemes for triggering LO independently of GC and show results for "best-in-class" variants in Figure 5. The *allocation rate* policy triggers a Generation 1 LO collection, if the allocation rate drops by more than 12.5%, and a Gen 2 LO collection, if it drops by more than 50%. The *DTLB+L2cache miss rate* policy triggers a Gen 1 LO collection when either the DTLB or L2 cache miss rate increases by 6.25%, and a Gen 2 LO collection, if either increases by 25%. The third policy is a combination of these two as described in Section 3.4. To summarize, it uses the allocation rate policy as the primary trigger and uses DTLB, L2 cache measurements when the allocation rate remains relatively stable. This combined policy provides better execution time benefits than the individual policies across all our C# applications as indicated in Figure 5. The final bar in Figure 5 indicates the impact of combining this triggering policy with our scheme for LO throttling (as described in Section 3.4). LO throttling does not decrease the benefits of our optimizations for any of the benchmarks. It is effective at reducing the performance degradation of GenIBC from -12% to -7%. In addition, it slightly increases our optimization benefit for Fugue by turning off LO for a brief period during its execution. All subsequent experiments use this triggering policy for LO (combined + throttling).

To ensure that the benefits do not arise from merely doing these additional GC at different times, we measured the effect of triggering Pro-active LO, but disabling the locality optimizations. We term this Pro-active GC and its impact is reported in Figure 6. As the figure indicates, it provides no execution time improvement and slows down a few of the programs by a small amount. The next bar in Figure 6 measures the traditional technique of combining LO with GC as done in prior research. Comparing this against Proactive LO (LO triggered independently of GC that uses the combined triggering policy with throttling as described above), indicates that in the cases where LO is effective, triggering it independently of GC provides large additional benefits. On average, for our set of C# applications Proactive LO improves execution time by 17% as opposed to almost no average improvement from traditional-style LO due to the slowdown this incurs for Max and GenIBC. If we ignore those two applications, Proactive LO improves performance by 27% whereas traditional LO provides 16% improvement. These results indicate that triggering LO independent of GC is effective. In addition, our LO throttling scheme limits performance degradation for programs with little algorithmic locality to just the profiling overhead.

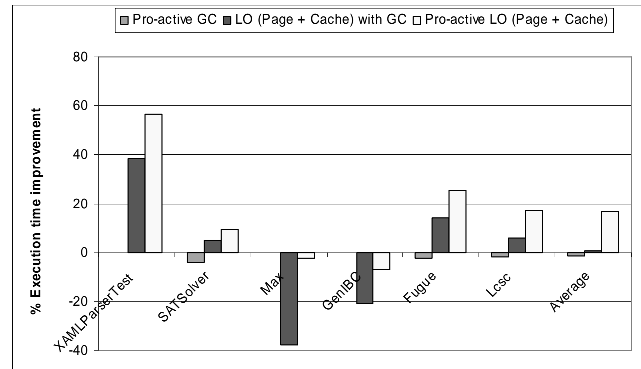

Figure 6: Benefits of triggering LO independently of GC.

4.3.3 Analysis of Pro-active LO Benefits

Finally, we performed experiments to separate out the locality benefits provided by page and cache optimizations. In addition, we measured locality metrics such as page density, data TLB misses, and L2 cache misses to validate that the observed execution time benefits arise from locality optimization.

Figure 7 indicates the execution time performance benefits of Proactive LO. The first bar represents the case where only page locality optimization is enabled. For the second bar, both cache and page locality optimizations are turned on. Page locality optimization produced improvements in XAMLParserTest and SATSolver and slowed down Fugue and Lcsc by a small amount. Overall, it improved execution time of our C# applications by 8% on average with a maximum improvement of 56% for XAMLParserTest. Combining this with cache locality produced additional improvements for SATSolver, Fugue and Lcsc for an overall average execution time improvement of 17%.

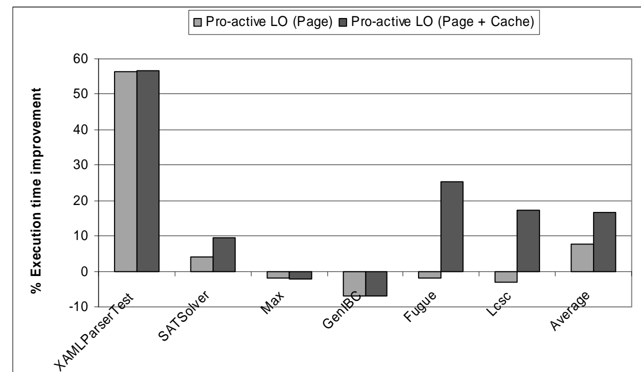

Figure 7: Page and Cache LO benefits for C# applications.

The results indicate the benefits of a system that combines page and cache locality optimizations. For some applications, page locality optimization is effective while others benefit most from cache locality optimizations. In addition, SATSolver benefits from both optimizations.

Table 2: Time spent in GC as a percentage of overall time.

Application	Base (%)	Pro-active LO (Page) (%)	Pro-active LO (Page+Cache) (%)
XamlParserTest	0.15	1	1.21
SAT solver	0.35	0.74	0.86
Max	26.0	26.7	26.9
GenIBC	27.0	27.5	27.6
Fugue	10.3	17.1	19.7
Lcsc	9.1	16.3	18.5

Table 2, which shows the fraction of execution time spent in the GC for different configurations, indicates that the execution time improvements arise solely from mutator speedup. The optimized configurations spend more time in GC to perform the locality optimizations, but these more than pay for themselves when effective.

Finally, Figure 8 shows the % reduction in page working set, data TLB misses, L2 cache misses, and % improvement in page density, for the Proactive LO (Page + Cache) configuration. All applications that benefit from our optimization incur a lower number of data TLB or L2 cache misses. For applications where

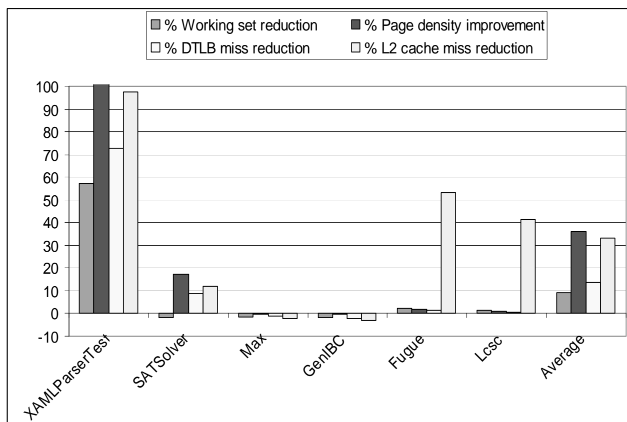

Figure 8: Locality Improvements for C# applications.

page locality optimization is effective (XAMLParserTest and SATSolver), the page density improves (by 196% for XAMLParserTest) and data TLB misses are reduced. For SAT solver, page working set increases slightly because the optimization involves many more GCs, each of which needs to scan portions or the whole heap, and these metrics do not exclude accesses made by the garbage collector. Applications that benefit from cache locality optimizations, such as SATSolver, Fugue, and Lcsc, show significant reductions in L2 cache misses (12—53%). XAMLParserTest is interesting in that the page locality optimization provides significant cache benefits as well. These numbers validate that locality optimizations are responsible for mutator speedups.

5. CONCLUSIONS

We have described an online profile-guided proactive approach to improve data locality in garbage collected systems. Our results show that it is beneficial to view the garbage collector as an explicit locality improvement mechanism rather than just a scavenger that is only invoked when the allocation budget is about to be exceeded.

We have shown that sampling can provide sufficiently detailed profile information to guide both page and cache locality optimization. Triggering LO independently of GC provides significant performance improvements over the traditional technique of performing LO with normal GC. Finally, combining page and cache locality optimizations in the same system provides larger benefits that either alone. These techniques improve the performance of the C# applications we studied by reducing both DTLB and L2 cache misses.

We are currently investigating further techniques for reducing the overhead of gathering profile data. A promising approach is to detect program phase changes [17][23]to guide the triggers for bursty sampling. We are also investigating the effects of different object field layout schemes for hot objects.

ACKOWLEDGEMENTS

We are grateful to Patrick Dussud for answering several questions pertaining to the CLR implementation. Hoi Vo and Hon Keat Chan offered advice on the implementation. Pramod Joisha, Ben Zorn, and the anonymous referees provided valuable feedback on earlier drafts of this paper.

REFERENCES

[1] Adl-Tabatabai, A., Hudson, R., Serrano, M., Subramoney, S. “Prefetch Injection Based on Hardware Monitoring and Object Matadata.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’04)*, 2004, 267—276.

[2] Arnold, M. and Ryder, B. “A Framework for Reducing the Cost of Instrumented Code.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’01)*, 2001, 168—179.

[3] Bacon, D.F., Cheng, Perry, Rajan V.T. “A Real-time Garbage Collector with Low Overhead and Consistent Utilization” In Principles of Programming Languages (POPL ’03), 2003.

[4] Cheney, C. “A Non-recursive List Compacting Algorithm.” *Communications of the ACM,* 13(11), November 1970, 677—678.

[5] Hirzel, M. and Chilimbi, T. “Bursty Tracing: A Framework for Low-Overhead Temporal Profiling.” In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization ’01 (FDDO)*, 2001, 117—126.

[6] Chilimbi, T. and Larus, J. “Using Generational Garbage Collection to implement Cache-conscious Data placement.” In *Proceedings of the 1st International Symposium on Memory Management*, October 1998, 37—48.

[7] Chilimbi, T., and Larus, J. “Cache-conscious Structure Definition.” In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI '99), 1999, 1—12.

[8] Courts, R. "Improving Locality of Reference in a Garbage-Collecting Memory Management System." *Communications of the ACM*, 31(9), September 1988, 1128—1138.

[9] Zorn, B. *The Effect of Garbage Collection on Cache Performance*, Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado at Boulder, 1991.

[10] Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Mateo, CA. 3rd edition, 2002.

[11] Hertz, M., Feng, Y., and Berger, E. D. "Garbage Collection without Paging" In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), 2005.

[12] Inagaki, T., Onodera, T., Komastu, H., and Nakatani, T. "Stride Prefecthing by Dynamically Inspecting Objects." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, 2003, 269—277.

[13] Lam, M., Wilson, P., and Moher, T. "Object Type Directed Garbage Collection to Improve Locality." In *Proceedings of the International Workshop on Memory Management*, 1992, 404—425.

[14] Hirzel, M., Diwan, A. and Hertz, M. "Connectivity-based Garbage Collection." In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03),* 2003, 359 – 373.

[15] Richter, J. "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework (Part I and II)." *MSDN Magazine*, 2000, http://msdn.microsoft.com/msdnmag/issues/1100/GCI/ and http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/.

[16] Moon, D. "Garbage Collection in a Large LISP System." In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, August 1984, 235 – 246.

[17] Nagpurkar, P., Krintz, C., and Sherwood, T. *Phase-aware Remote Profiling*, Technical report UCSB 2004-21, Department of Computer Science, University of California at Santa Barbara, 2004.

[18] White, J. "Address/Memory Management for a Gigantic LISP Environment or, GC Considered Harmful." In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, 1980, 119 – 127.

[19] Wilson, P. "Uniprocessor Garbage Collection Techniques." In *Proceedings of the International Workshop on Memory Management*, 1992, 1 – 42.

[20] Wilson, P., Lam, M., and Moher, T. "Effective Static-graph Reorganization to Improve Locality in Garbage Collected Systems." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, 1991, 177 – 191.

[21] Shuf, Y., Gupta, M., Franke, H., Appel, A., and Singh, J. "Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times." In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02),* 2002, 13 – 25.

[22] Huang, X., Blackburn, S., McKinley, K., Moss, J., Wang, Z., and Cheng, P. "The Garbage Collection Advantage: Improving Program Locality." In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04),* 2004, 29 – 80.

[23] Shen, X., Zhong, Y., and Ding, C. "Locality Phase Prediction." In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, 2004, 165 – 176.

[24] Chilimbi, T. "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality." In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, 2001, 191 – 202.

[25] Wilkes, R. "Ngen Revs up your performance with Powerful New Features", MSDN Magazine April 2005