Gated Memory Control for Memory Monitoring, Leak Detection and Garbage Collection

Chen Ding, Chengliang Zhang, Xipeng Shen, and Mitsunori Ogihara

Computer Science Department
University of Rochester
{cding,zhangchl,xshen,ogihara}@cs.rochester.edu

1 Abstract

In the past, program monitoring often operates at the code level, performing checks at function and loop boundaries. Recent research shows that profiling analysis can identify high-level phases in complex binary code. Examples are time steps in scientific simulations and service cycles in utility programs. Because of their larger size and more predictable behavior, program phases make it possible for more accurate and longer term predictions of program behavior, especially its memory usage. This paper describes a new approach that uses phase boundaries as the gates to monitor and control the memory usage. In particular, it presents three techniques: memory usage monitoring, object lifetime classification, and preventive memory management. They use phase-level patterns to predict the trend of the program's memory demand, identify and control memory leaks, improve the efficiency of garbage collection. The potential of the new techniques is demonstrated on two non-trivial applications—a C compiler and a Lisp interpreter.

2 Introduction

Dynamic memory allocation is used in most software systems. It comes with the problems of program errors, caused by deallocating a live object, and memory leaks, caused by not deallocating a dead object. The first problem is often solved by conservative memory recollection, which only excerbates the second problem of memory management. In this paper, we study the problem of both manual and automatic memory management.

Decades of research in memory management has produced highly effective methods for managing dynamic objects. Because programs are complex, the existing methods take a general program model and analyze at the level of procedures and allocate sites. Recently, program phase analysis makes it possible to automatically identify high-level phases that span thousands lines of code. The goal of our work is to understand how much the high-level phase information can benefit memory management.

We define a phase as a unit of recurring behavior, and the boundaries of a phase can be uniquely marked in its program. We have recently introduced a profiling-based method [17]. It uses small-scale profiling runs to find common temporal patterns. The key concept is reuse distance. By examining the reuse distance of varying lengths, the analysis can "zoom in" and "zoom out" over long execution traces. It detects phases using signal processing and program analysis techniques. It identifies composite phases and represent them as a hierarchy. Finally, it inserts marker instructions into the application via binary rewriting.

We recently improved phase analysis with a new technique called *active profiling* [16]. It exploits the following observation: if we provide a utility program with an artificially regular input, then its behavior is likely to be regular as well. This regularity allows us to identify recurring behavior and delimit the high-level processing steps during normal operation as well, when irregular inputs make traditional analysis infeasible. The automatic analysis has found service cycles in utility programs, including a data compression program, a compiler, an interpreter, a natural

language parser, and a database. In this paper, we evaluate our techniques on two of these programs—the C compiler *GCC* and the Lisp interpreter *xlisp*.

The gated memory control is to perform various techniques at the phase boundaries. An example of gated memory control is shown in Figure 1. The example program handles a sequence of incoming requests. The memory control is applied between two consecutive requests. Each request is processed in two steps. An inner gate can be inserted at the beginning or the end of these sub steps. In this paper, we describe three schemes for gated memory control for outermost phase instances.

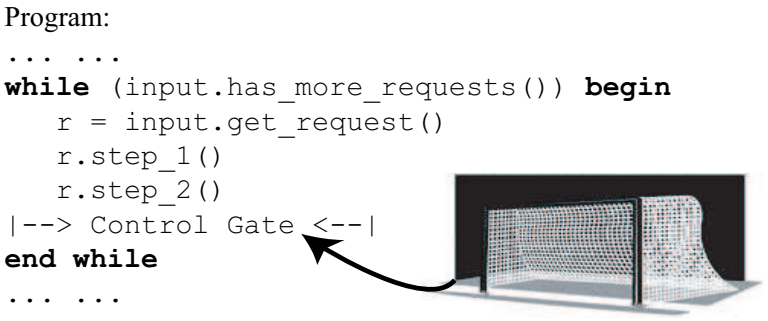

Figure 1: Gated memory control based on the phase structure of a program

The first technique is *memory usage monitoring*. It measures the heap size at the end of outermost phase instances. Since objects that live across multiple phase instances tend to remain in the heap permanently, the boundary check provides an accurate measure on the long-term trend of the memory usage. The trend is not as visible without phase-based checking. For example, a program processes a sequence of requests. Each service may take a different amount of memory during the service. Without the phase structure, a monitor would not be able to measure the memory use at the same stage of each service.

The second technique is *object lifetime classification*. We define usage patterns that separate those objects that are used only inside a phase instance from those that are shared by multiple phase instances. We classify the allocation sites and identify possible sources of memory leaks. In addition, we identify objects that are not used until the end of the execution. The classification and pattern analysis has the potential to identify hard-to-find memory leaks, discover reachable dead objects, and help to reduce the size of live data.

The last technique is *preventive memory management*. Unlike the traditional reactive schemes, the preventive method applies garbage collection (GC) at the beginning of each phase instance and let the heap grow unimpeded during a phase instance. Based on the high-level program structure, it performs GC at the right time so that dynamic memory is allocated based on the natural need of an application.

Phase analysis inserts static markers into the program binary without accessing the source code. It does not rely on high-level program information such as loops and procedures and can handle programs where the control and data structures are obfuscated by an optimizing compiler. Therefore, gated memory control can be applied to a wide range of applications including open-source utilities, legacy code, and dynamically linked programs. In the rest of the paper, we describe these three techniques and our preliminary experimental results.

3 Gated memory control

3.1 Monitoring the trend of memory demand

A phase, especially the outermost phase, often represents a memory usage cycle, in which temporary data are allocated in early parts of a phase instance and then recollected by the end of the instance. Some data may live across phase instances. For a long running program with many phase instances, the amount of data shared by multiple phase

instances determines the trend of the memory demand. For example, *GCC* may allocate many temporary objects during the compilation of an input function, deallocate most of them, and keep only the information needed by the compilation of succeeding functions.

If the long-term memory usage is mostly determined by the increase in the objects that live across multiple instances of the outermost phase, we can monitor the trend of the memory demand by measuring the size of these objects. For programs that use manual memory recollection, including pool-based deallocation, the monitoring technique instruments the memory allocator to record the current size of allocated data. At the end of each outermost phase instance, the size of the heap is recorded. As a program executes, it uses the sequence of heap sizes to identify the trend of memory usage.

We have applied this monitoring technique on one execution of *GCC*, whose usage trend is shown by three figures. The upper curve in Figure 2 is the heap sizes measured at the end of the phase instances. It shows that on average, the heap size grows by 58KB per instance. The long-term memory demand increases steadily. The trend is not obvious without phase-based monitoring. For reasons of curiosity, we also measured the size of the data that have a later use at the phase bounary. The trend is shown by the lower curve in Figure 2. It is relatively flat compared to the curve of the actual heap size. The first curve in Figure 3 shows the size of the heap at the end of every 3 million memory accesses. The usage trend is skewed by the length variation of different phase instances. The second curve shows the size of the data that have a later use, which does not show a consistent trend.

Figure 4 shows the IPC trace of the execution on IBM Power4, with vertical bars marking the boundary of phase instances. Most execution time is dominated by a handful of the 107 phase instances. Therefore, the measurement in Figure 3 comes mostly from large phase instances. Even if one measurement is taken once for each phase instace, there is still no guarantee of correlation if we cannot measure at the same stage of each phase instance. Therefore, we need the phase information to monitoring the long-term memory usage.

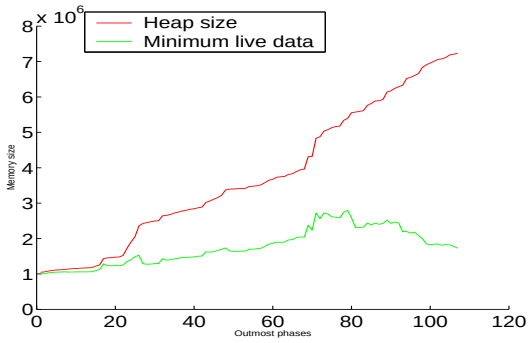

Figure 2: The memory usage trend over the instances of the outermost phase

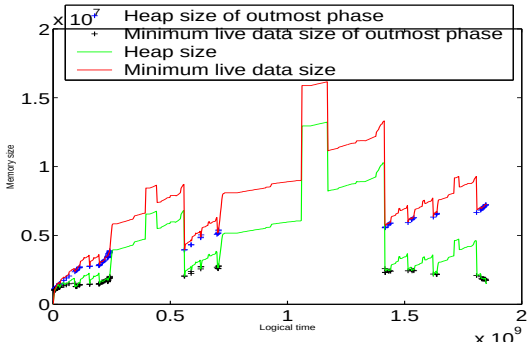

Figure 3: The size of the heap at the end of every 3,000,000 memory accesses

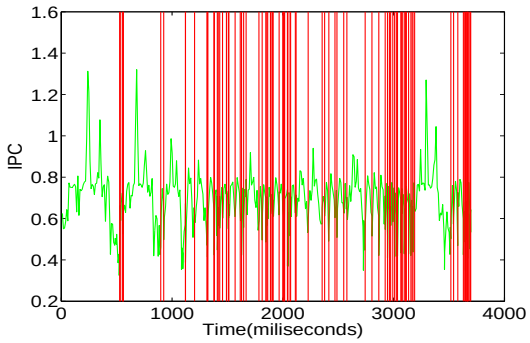

Figure 4: the memory usage trend over the instances of the outermost phase

The same monitoring technique can be used on a garbage collection system by invoking GC at the end of each phase instance and measure the size of reachable data. The overhead is potentially high because the program may perform more GC than needed. However, as the later section will show, applying GC regularly at the end of phase instances actually improves performance.

The memory usage monitoring improves program reliability perceived by users. They want to know whether a program can finish serving the current requests without exhausting limited memory resource. The memory usage is becoming more important as programs are entering embedded systems. Today a cellular phone can handle email, browse the Web, take photos, record audio and video clips. The monitoring technique helps a user to predict the amount of memory needed by a program on a given input. It can warn a user of insufficient resource before the failure happens. The information is important not just for long running programs. Some programs may take a small input most of the time and large inputs only occasionally. Consequently, if an insufficient-memory error comes, it comes completely unexpected and often at a time of a critical need by the user.

3.2 Object lifetime classification

The phase structure of an application allows us to model and measure memory usage patterns. The life-time of a dynamic object forms an event. The events are grouped by their run-time allocation sites (labeled by the run-time call stack [3]). We mark the beginning and ending times of an event by the phase time. We will identify common patterns through profiling. For example, some events begin and finish in the same phase instance, and some events begin and finish in adjacent instances. We identify major patterns for each event group. Then events that do not confirm to the majority pattern are candidates for further scrutiny.

We classify dynamic objects as either *phase local*, if their first and last accesses are within an outermost phase instance, *hibernating*, if they are used in a single phase instance and then have no use until the last outermost phase instance of the execution, or *global* if they are used by multiple outermost phase instances. The phase time differs from the physical time. Since phase instances may have very different execution times, a phase local object in a long phase instance may have a longer physical life than a global object.

In the preliminary study we analyze two types of patterns. The first indicates a possible memory leak. If a site allocates only phase-local objects during profiling, and if not all its objects are freed at the end of the phase instance, then it is likely that the remaining objects are memory leaks. However, if some of the allocated objects have a global use, then the unfreed phase-local objects may not be memory leaks. The program may keep it around for possible later uses that do not happen in this particular execution.

The second pattern we examine is the hybernating objects. If a site allocates only hybernating objects, then it is likely that we can avoid storing the object for more than one phase instance either by moving the last use early or by storing the object in disk and reloading it at the end. In addition, we will group objects that are unlikely to be used during the same period of time and place them into the same virtual memory page. Object grouping does not reduce

the size of live data but it reduces the amount of physical memory usage because the unused page can be stored to the disk by the operating system. Object grouping may also reduce the energy consumption without degrading performance when their memory pages are placed in a sleeping mode.

Through profiling, the analysis can identify all allocation sites that contribute to the long-term memory increase. We can rank them by the rate of their contribution to the memory increase, so a programmer can fix the most pressing problems and leave the mild ones to the run-time monitoring system.

The profiling methods may not detect a memory leak if it does not happen during profiling. In general one cannot guarantee a program free of memory leaks. We can expand the coverage by using multiple training inputs and by constructing special inputs. When a memory leak passes through all the checks but still surfaces in an execution, we can still provide a early warning to a user, as described in the previous section.

We have implemented a prototype classifier and tested it on *GCC* over a number of large inputs. We next show two example results from the report generated for the input 200.i. This version of *GCC* uses manual allocation and free operations to manage dyanmic data.

The first is the allocation site that accounts for the largest increase in global data. A generated report is given below. It shows that a single site allocates 924 objects for a total of 3.8 million bytes. None of the allocated objects is freed during the execution. They account for the largest portion of the uncollected data in an execution of *GCC*. Based on the function names in the stack trace (the integer is the unique identifier for each static call site in the program), the objects appear to be information stored for use in later inlining. The usage pattern shown in the second part of the report is consistent with this conjecture. Most of the objects are not used by later phase instances but some are. We tested six inputs—the entire reference and training inputs of the benchmark. The objects allocated by this call site and another allocation site are ranked in the top two in all the inputs.

```
alloc. site: 44682@xmalloc<149684@_obstack_newchunk<149684@rtx_alloc<
             183998@save_for_inline_copying<46828@rest_of_compilation<
             23232@finish_function<890@yyparse<45674@proc_at_0x12005c390<
             48606@main<390536@__start<

924/3762528 unfreed, 0/0 freed.
                              freed              unfreed
    phase local         0/         0      875/   3563000
   hiberanating         0/         0        0/         0
         global         0/         0       49/    199528
```

The next example report shows another site, which allocates 18 4KB objects. The program recollects 14 of them but the other 4 are not recollected. Since all 18 objects are phase local in this execution, it is possible that the 4 remaining objects are memory leaks. In our future work we will examine cases like this to better understand reasons for this usage pattern.

```
alloc. site: 44682@xmalloc<149684@_obstack_newchunk<149684@rtx_alloc<
             155387@gen_rtx<352158@gen_jump<84096@proc_at_0x120082260<
             83994@expand_goto<4308@yyparse<45674@proc_at_0x12005c390<
             48606@main<390536@__start<
4/16288 unfreed, 14/57008 freed.
                              freed              unfreed
    phase local        14/     57008         4/     16288
    hibernating         0/         0         0/         0
         global         0/         0         0/         0
```

In all inputs of *GCC*, we found that all objects recollected by the program are phase local objects. Once an object escapes recollection in the phase instance of its creation, it will not be recollected until the end.

3.3 Preventive memory management

For programs that use automatic garbage collection, we have envisioned a scheme we call *preventive memory management*. The new scheme applies garbage collection at the beginning of each outermost phase instance. In addition, it does not apply garbage collection in the middle of a phase instance unless the heap size reaches the hard upperbound on the available memory. It differs from existing garbage collection schemes that are invoked when the heap size reaches a soft upperbound. The GC would first recollect unreachable objects and allocate more memory if the free space after GC is below a threshold. We call the latter scheme *reactive memory management*. There are three reasons that a preventive scheme may perform better than reactive schemes:

- The preventive scheme performs fewer GC passes.
- It allows the heap to grow unimpeded inside a phase instance.
- The data locality may be improved due to preventive GC.

Preventive GC takes a middle point between two extremes. One is not to do garbage collection unless the program runs out of a hard memory constraint, for example, the 32-bit address space. If the total amount of dynamic data is less than the hard limit, then the execution can finish without any garbage collection. The problem with this scheme is that the program occupies a large amount of memory space throughout the execution. A conservative garbage collector takes the other extreme. It increases the avaiable memory by a small amount at time, and it always runs GC before an increase (so to avoid the increase whenever possible). The total memory usage is kept at minimium, but the overhead is high because of repeated GC in the middle of large phase instances.

The modern garbage collection schemes have explored various parts of the space between the two extremes. For example, the generational garbage collector separates old from new objects and reduces the cost of each GC pass. However, various thresholds are needed to determine the classfication and to accommodate varations in typical memory loads. This and other solutions are run-time based but do not exploit the high-level program behavior pattern. The preventive GC provides an interesting alternative. In the basic case, it does not need any thresholds other than hard memory upperbound. It is a unique way of trading off program speed and memory usage.

As an initial idea, preventive GC is far from practical. It is not clear how it interacts with modern, more sophisticated garbage collection schemes. Neither is certain whether it can be implemented on current and future virtual machines. However, the idea warrants attention because it opens a way to combine high-level program information with dynamic memory management. In standalone applications that use a conservative garbage collector, the current preventive scheme is applicable. Next we evaluate preventive GC for a non-trivial application.

The program *xlisp* is a Lisp interpreter from the SPEC95 benchmark set. The original garbage collector works as follows. All dynamic data are list nodes stored in segments. It keeps a list of free nodes and allocates from the list as the request comes. Once the size of free list drops below a threshold, it invokes garbage collection. If the size of free list does not rebound back, it allocates one segment. Each segment contains 1000 nodes.

We implemented the preventive garbage collection and tested the performance on an Intel Pentium 4 workstation. We used both the train and the ref inputs. The execution time of the entire program on Intel Pentium 4 is shown in Table 1. Using preventive GC, the program outperforms the version using reactive GC by 44% on the Intel machine for the reference input and and a factor of 3 for the train input.

The faster execution time is due mainly to fewer GC passes. Preventive GC passes are 3 times fewer than reactive ones for the train input and 111 times fewer for the reference input. Another possible reason is better locality. We could not measure locality directly in this experiment but we plan to use hardware counters to record the number of cache misses in executions of the program and the GC sub-part.

Because the preventive scheme does not perform garbage collection during a phase instance, it may consume much more memory than it would under the reactive scheme. However, the high memory usage is temporary. After a phase instance, the preventive GC will reduce the memory usage back by recollecting all phase local objects. The garbage collector of *xlisp* does not copy and compact live data. It does not free memory once it is allocated. The

program inputs	GC methods	exe. time (sec) Pentium 4	per-phase heap size (1K)		total GC calls	total seg. visited by GC
			max seg.	avg seg.		
ref.	preventive	13.58	16010	398	281	111883
	reactive	19.5	17	11	31984	387511
	no GC	12.15	106810	55541	0	0
train	preventive	0.02	233	8	47	382
	reactive	0.07	9	3	159	1207
	no GC	0.03	241	18	0	0

Table 1: Comparision of the excution time between preventive and reactive GC

recollected nodes are stored in a free list. We modified the program to free the memory when all nodes in a segment are free. As shown by the two middle columns in the table, less frequent garbage collection leads to larger heap sizes. We take the average of the maximal heap size of all phase instances as the average size of the heap for a phase instance. The average size using the preventive GC is 2.7 times larger than the reactive case for the train input and 36 times larger for the reference input.

We have tested the ideal scheme where no GC calls are made. We found that the performance of the preventive scheme is close to the ideal scheme. These results suggest that the conservative method of applying GC before increasing available memory has a high performance overhead. Using the phase information, the preventive scheme recollects memory at the exact time when all phase-local data are no longer useful. It does not interfere in the middle of a phase instance, letting the heap size grows with the program's need. We believe that the reactive scheme can be improved to allow bursty allocation and avoid repeated recollection when the number of objects freed is low. Previous work may have already done this. There must be some design of the garbage collector that can obtain a similar performance as we have. However, the uniqueness of the preventive scheme is its simplicity. It does not use highly tuned thresholds or parameters. Instead, it is based on the high-level program behavior pattern.

4 Related work

Manual allocation and deallocation are used in most software systems. The allocator performance has steadily improved [4, 19]. Increasingly garbage collection is used to replace unsafe manual deallocation. Garbage collection finds and recollects unreachable objects. Open-source implementations have been widely distributed for C/C++ [5], Java, and CLR. Memory management is an active area of research and has a direct impact on not just the memory footprint but also the memory performance [7, 13, 20]. In this work we explore the use of gated memory control to improve memory management. Our preliminary results show the potential of memory reduction by more than a factor of three over highly hand-crafted *GCC* program. A limit study shows that the exact live information would lead to a similar improvement over automatic garbage collection in Java and Effiel programs [11]. The high-level phase structure will enable pattern analysis, which in turn may provide high-level feedback and more precise diagnoses. In addition, gated resource control provides early warnings for insufficient-memory failures. It can reduce the size of live data, as suggested by our preliminary results. Finally, it may improve the performance of garbage collection by applying it at more appropriate times during an execution.

Chilimbi and Hauswirth used efficient run-time monitoring and statistical pattern analysis to identify as likely memory leaks the data that are not accessed for an extended period of times [6]. Our detection scheme is based on profiling, but the phase time would be equally useful in the dynamic detection method of theirs.

Program phases can be analyzed by examining the high-level program constructs such as loops and functions [2, 14, 15] and regions [1, 12] or by finding patterns among the run-time intervals of an execution using hardware support [2, 8, 9, 10, 18]. Our earlier work showed that binary profiling can be used to identify high-level program phases by combining data, code, and run-time analysis [16, 17]. We use high-level phases in gated memory control.

5 Summary

This paper describes a new approach that use phase boundaries as the gates to monitor and control the memory usage. In particular, we have presented three techniques: memory usage monitoring, object lifetime classification and preventive memory management. Memory usage monitoring measures the heap size at the end of the outermost phase instances. It help improve program reliability by showing whether a program can finish serving the requests without exhausting the limited memory resources. Based on dynamic memory usage patterns in term of phase distance, we identify three classes of dynamic objects: phases local, hibernating, or global. We also rank the memory sites that contribute to long-term memory increase. These information is potentially valuable for the programmer to fix the most pressing problems. Preventive memory management apply garbage collection at the beginning of outermost phase instance. Experiments show that the preventive method improves performance by over 40%, compared to a version of traditional reactive memory management.

Acknowledgments

This material is based upon work supported by a grant from The National Science Foundation grant numbers E1A-0080124 and EIA-0205061 (subcontract , Keck Graduate Institute), and Department of Energy award #DE-FG02-02ER25525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

References

[1] F. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1976.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Monterey, California, December 2000.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2002.

[5] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, September 1988.

[6] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA, October 2004.

[7] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the International Symposium on Memory Management*, 1998.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working-set analysis. In *Proceedings of International Symposium on Computer Architecture*, Anchorage, Alaska, June 2002.

[9] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of International Symposium on Microarchitecture*, December 2003.

[10] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.

[11] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 2002.

[12] C.-H. Hsu and U. Kremer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[13] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2004.

[14] M. Huang and J. Renau and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[15] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, , and S. Dropsho. Profi le-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[16] X. Shen, C. Ding, S. Dwarkadas, and M. L. Scott. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.

[17] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

[18] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[19] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: a survey and critical review. In *Proceedings of the International Workshop on Memory Management*, 1995.

[20] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.