Online Optimizations Driven by Hardware Performance Monitoring

Florian T. Schneider
Department of Computer Science
ETH Zürich
Zurich, Switzerland

Mathias Payer
Department of Computer Science
ETH Zürich
Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zürich
Zurich, Switzerland

Abstract

Hardware performance monitors provide detailed direct feedback about application behavior and are an additional source of information that a compiler may use for optimization. A JIT compiler is in a good position to make use of such information because it is running on the same platform as the user applications. As hardware platforms become more and more complex, it becomes more and more difficult to model their behavior. Profile information that captures general program properties (like execution frequency of methods or basic blocks) may be useful, but does not capture sufficient information about the execution platform. Machine-level performance data obtained from a hardware performance monitor can not only direct the compiler to those parts of the program that deserve its attention but also determine if an optimization step actually improved the performance of the application.

This paper presents an infrastructure based on a dynamic compiler+runtime environment for Java that incorporates machine-level information as an additional kind of feedback for the compiler and runtime environment. The low-overhead monitoring system provides fine-grained performance data that can be tracked back to individual Java bytecode instructions. As an example, the paper presents results for object co-allocation in a generational garbage collector that optimizes spatial locality of objects on-line using measurements about cache misses. In the best case, the execution time is reduced by 14% and L1 cache misses by 28%.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Optimization, Runtime Environments, Memory Management

General Terms Measurement, Performance

Keywords Java, Just-in-time Compilation, Dynamic Optimization, Hardware Performance Monitors

This research was supported, in part, by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation, and by a gift from the Microprocessor Research Lab (MRL) of Intel Corporation.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

1. Introduction

Object-oriented programming languages like Java or C# allow changes to an executing program at runtime, e.g., through the use of a dynamic class loader. At the same time, modern processor architectures are difficult compiler targets if the compiler aims to optimize a program for speed of execution; features like prefetching and prediction are (sometimes) difficult to model in a compiler. So a code generator is faced with two difficulties: the dynamic nature of the target program complicates analysis of program properties (e.g., it is difficult to determine pointer aliasing or to analyze the memory referencing patterns) and important performance aspects (e.g., number and location of cache misses) are only evident at runtime.

Fortunately, programs written in such an object-oriented language are usually executed in a virtual machine that includes a JIT (dynamic) compiler. The dynamic compiler has the opportunity to immediately make use of information obtained at runtime. We distinguish between two kinds of information about an application that can be obtained at runtime:

- information that is independent of the execution platform like the execution frequency of methods, basic blocks or instructions; often the term *profiles* is used for this kind.
- machine-level information, i.e. performance data about the "lower levels" of the execution platform (OS, hardware). Examples for this type of information are cache misses, TLB misses, or branch prediction failures.

Profiles are a useful input to the code generator (not only in a JIT compiler but also in an ahead-of-time compiler). However, many previous optimizations (static and dynamic) focus only on the platform-independent information and did not include direct feedback from the hardware level [23, 20]. Yet most modern CPUs (like the Pentium 4, Itanium, PowerPC) have a performance measurement unit to obtain performance-related information and therefore could provide input to a dynamic code generator that optimizes a program for a specific hardware platform.

To be useful for an optimizing JIT compiler and associated runtime system[1] the collected performance information must be *accurate* enough and *cheap* to obtain at run-time. There are a couple of requirements for a module that makes information from the hardware performance monitors available in such an execution environment:

- The interface between the VM and the performance monitoring hardware should hide machine-specific details where possible.

[1] We consider the JIT compiler, the virtual machine (VM), and the runtime system as one unit since all components must cooperate to perform most interesting optimizations.

It should be flexible to allow obtaining different execution metrics.

- The overhead to collect the data should be as low as possible, and the system should not perturb executed applications too much.
- The information must be accurate enough to be useful for online optimization. Often the granularity of a method or even a basic block is too coarse to infer which operation is responsible for some event (e.g. cache misses).
- The platform should work for off-the-shelf VMs, with only small or no changes to the core VM code. Otherwise the effort to port the infrastructure to another VM or to a new release would be prohibitively large.

In this paper we show how a Java system can benefit from using machine-level performance data, but the approach and results are not tied to the Java programming language. Of course, any compiler that uses platform-specific information may also use profile information, e.g., to decide where and when to exploit the results obtained from the performance measurement unit, but this aspect is not discussed further in this paper. We describe and evaluate a module to feed data from the hardware performance measurement unit of a modern processor into the Java system. Our infrastructure is built on top of the Jikes RVM, a freely available open-source research VM implemented in Java. In our system we exploit special features of the P4 processor that allow to correlate measured events to single instructions and to the source program (in our case Java bytecode). The overhead of the runtime hardware sampling is reasonably low ($<$1% avg).

As an example application of our infrastructure we present a garbage collector that is guided by online hardware feedback and report the results for a selection of standard Java benchmarks. The garbage collector improves data locality of Java programs automatically by co-allocating heap objects using information about data cache misses. The principal idea is to identify those objects and references that "produce" the largest number of cache misses. The garbage collector uses these hints to adapt its behavior for better data locality. Our systems is, however, not aimed just at data locality optimizations in the GC. Instead machine-level performance data should be thought of an additional feedback for the whole runtime environment. We chose this optimization to demonstrate that the overhead of the approach is low in practice to allow a code generator/runtime system to deal with memory performance – one of the difficult areas for a compiler for object-oriented programs.

The next section discusses prior work in the area. Section 3 shortly presents the hardware and software platform we used. We present an adaptive runtime monitoring system in more detail in Section 4. Section 5 describes an example application of our system on data locality optimization during GC. Finally, we discuss the overhead and the impact on performance in Section 6. The key contribution of this paper is that it is actually possible to collect detailed data from the hardware during runtime that can be directly applied for optimization by the runtime environment.

2. Related work

There are two areas of prior work that we discuss in this paper: data gathering techniques using profiles or hardware performance monitors (HPMs) and data locality optimizations. For data gathering techniques we focus on approaches for dynamic compilation and optimization. There exists a fair bit of prior work about profiling and profile-guided optimization in ahead-of-time compilers (see, e.g., [21, 12]) that is however not central to the topic of this paper.

Hardware performance counters are frequently used for off-line performance analysis and characterization of workloads. Hauswirth et al. [16] study the interaction between the VM and the lower levels of the execution platform (OS, libraries, hardware). They measure how these layers influence each other by introducing "software performance counters" which capture performance metrics of the software subsystems and correlate them to the information gathered by the hardware performance counters.

To correlate data from the hardware with Java methods Georges et al [15] instrument method entries and exits with reads of the hardware performance counters. Their approach reduces the number of instrumentations significantly by first identifying execution phases and then only instrumenting the start and the end of these phases. This way the high overhead of instrumenting every method can be avoided.

Several high performance JVMs use adaptive optimization based on run-time profiling [4, 14, 27]. The Jikes RVM [7] uses timer-based sampling of the call stack to find frequently executed methods. The frequency profiles are used to determine where to spend the most effort for optimization: The more often a method is invoked, the more expensive optimizations are applied to it. A static cost/benefit model for the different optimizations is used to evaluate whether a method should be recompiled. It also has the ability to use continuous profiling feedback to improve performance of long-running applications [8].

Several studies show that data locality optimizations can improve the performance of programs with irregular memory access patterns. Field reordering [19, 22] is a technique that targets objects that do not fit into one cache line. It places fields with high temporal affinity together to improve cache utilization. Class splitting [13] achieves a similar effect by splitting data structures into two: a hot (frequently accessed) and a cold part. The hot parts are allocated together to avoid infrequently used data to use up cache memory. Usually these techniques rely on profiling information to approximate a good data layout because it is generally hard to statically optimize data locality in object-oriented programs.

Adl-Tabatabai et al. [3] present a dynamic optimization to eliminate long-latency cache misses. They insert prefetch instructions after dynamically monitoring cache misses using hardware performance monitors of the Itanium processor. The approach exploits the fact that objects that are linked through a reference often have a constant delta between their starting addresses. Software prefetching must be used consciously because fetching the wrong data into the cache may have a negative performance impact. By using hardware performance monitors to guide the prefetching they achieve a speedup of 14% for the SPEC JBB2000 benchmark. In our application we do not target prefetching, but instead we reorder object instances to reduce the number of cache misses.

Online object reordering [17] is a different dynamic locality optimization for Java. It reorders objects at garbage collection time using a copying GC. The heuristic for reordering is determined by profiling the field access operation with a light-weight profiling mechanism. Objects with "hot" fields (frequently accessed) are placed adjacent to their referent objects to increase spatial locality by visiting those references first during the copying process in the GC. and reduce the number of cache misses. The approach requires a copying garbage collector (which is present in many modern VMs). Our work takes a similar approach, but we do not rely on execution frequencies as a metric for locality. Instead we use direct feedback from the memory hierarchy about cache misses to guide compiler and GC decisions.

Similar ideas have been used to improve code locality. Dynamic code management [18] is a code reordering algorithm to improve code locality and reduced ITLB stalls. The system builds up a call graph at runtime and uses a light-weight reordering heuristic to

determine the optimized code layout which results in a speedup of up to 6%.

Shuf et al [24] also use the memory management system to improve data locality and present an object allocation scheme that attempts to place frequently instantiated types that are connected via references close together in memory. They also show that a locality-based heap traversal algorithm can improve GC performance.

Lau et al [20] show how to use direct measures of performance (cycle counts) to guide inlining decisions in a dynamic compiler. The JIT generates two version of each method: one with aggressive inlining and one with the default (more conservative) heuristic. By executing each of the two versions randomly during the measurement phase the compiler collects timing information about each version. After filtering out outliers it can use those timings to decide which version of the method to use in future. Our approach also uses real machine metrics as feedback, but gathers more fine-grained information about the program's interaction with the execution platform (like cache or TLB misses).

3. Background

3.1 P4 hardware performance monitors

The P4 offers a large variety of performance events for counting [2, 26]. Two modes of operation are supported:

- Normal counting: The performance counters are configured to count events detected by the CPU's event detectors. A tool can read those counter values after program execution and reports the total number of events. This mode can be used to obtain numbers like cache miss rate, total execution cycles, etc.) More fine-grained information (e.g., on a method level) can be obtained by instrumenting the program for reading counter values. An application of this mode would be to evaluate the precise effect of program transformations.
- Sampling-based counting: Whenever a certain number of events has occurred, the CPU samples its register contents. This way it is possible to locate the source of an event. The P4 supports precise event-based sampling (PEBS), i.e., it reports the exact instruction where the sampled event happened and the register contents at that point although the design is heavily pipelined. Previous CPUs could only measure an approximate location for sampled events because of a super-scalar design and out-of-order execution.

  To reduce the overhead of sampling, the CPU has a special microcode routine that saves the CPU state to a buffer supplied by the OS whenever the interval counter triggers the sampling of an event. Setting the interval counter to n means that every nth event will be sampled.

3.2 Jikes RVM

Our implementation is done with the IBM Jikes RVM (version 2.4.2) [5, 4], a high performance Java virtual machine written mostly in Java. It includes an adaptive optimization system (AOS) [7]: First, every method is compiled with a simple and quick baseline compiler. To estimate the execution frequency for methods it samples the call stack in regular intervals and records which methods are on top of the stack. Methods that are executed frequently enough are recompiled and optimized further. The VM uses a static cost model to decide which optimization level to apply for a method.

4. Infrastructure

4.1 Event sampling

We use the precise event-based sampling (PEBS) feature of the P4 processor [2] to measure cache misses. This mechanism has two advantages that make it useful for monitoring applications during runtime: First, the CPU collects event samples on its own using a microcode routine and stores them into a buffer supplied by the OS kernel module. An interrupt is generated only when this buffer is filled to a specified mark. The second advantage is that PEBS reports the exact instruction (program counter plus all register contents) for the sampled events. This allows the compiler to recover higher-level information about the collected events, e.g., method, bytecode instruction, or field variable accessed. The P4 has a number of events that can be selected for PEBS (e.g. L1, L2 cache misses and DTLB misses), but it allows only one event to be measured at a time.

The system consists of three parts, see [23] for an overview and general description:

1. Perfmon loadable kernel module [1][2]: This kernel module is part of the Perfmon infrastructure and is developed at HP. It offers the functions to access the performance counter hardware for a variety of hardware platforms. The kernel module hides the platform-specific details from the JVM. It also provides the interrupt handler that is called by the sampling hardware when the CPU buffer for the samples is full.
2. Native shared library (C): Since we cannot call device drivers directly from Java or from the Jikes RVM we developed a native library to provide an interface to the kernel functions and access it via the Java Native Interface (JNI). The library allows to read samples from the kernel module. The challenge here is to make the data exchange between Java and the kernel as efficient as possible. We provide a pre-allocated array to the native code. The library function then copies all collected samples into this array directly without any JNI calls. We only need to make sure that the GC does not interfere during this transfer. This can be done by disabling the GC for the short period of time while the samples are copied from the native library. Since no allocation happens in the native code that is responsible for copying, we can make sure that the GC is not triggered while samples are copied from the kernel space.
3. Collector thread (Java): We use a separate Java thread that polls the kernel device driver via the JNI interface whether there are any new samples. The polling interval is adaptively set between 10ms and 1000ms depending on the size of the sample buffer and the sampling rate. This makes sure that no samples will be dropped due to a full sample buffer.

The copying of samples into user-space is necessary to allow the use of a different hardware platform with very few changes to the user-space library. The library is not limited to Java and can also be used from other runtime environments and languages. Basically the system can also be integrated with other Java VMs that support JNI.

4.2 Mapping HPM data to Java bytecode

One sample on the P4 platform has a size of 40 bytes. It contains the program counter (EIP) where the sampled event occurred and the values of all registers at that time. At the moment we do not monitor the data register contents and only analyze the EIP register. To actually use the raw data for optimization, we need to obtain higher-level information about each sample:

[2] available for download at http://www.hpl.hp.com/research/linux/perfmon/

First the collector thread extracts the samples that are of importance for the VM. Addresses outside the VM address space (e.g., from kernel space or native libraries) are dropped immediately since we are only interested in events that occur in the machine code generated by the JIT compiler. The next step is to find the Java method where the event happened. For this lookup we keep a sorted table of all methods with their start and end address. Whenever a method is compiled the first time or recompiled by the optimizing compiler we update its entry accordingly. For simplicity, code for compiled methods is allocated in the immortal object space of the VM which is not garbage-collected. This way the copying GC does not move compiled code which would require an update of the lookup table after every GC run (up to 18K method objects in our benchmarks). The resulting space overhead due to stale method objects is however reasonably small because in our setup only a small fraction of methods are re-compiled and replaced by the optimizing compiler.

Finally the system determines the exact bytecode instruction for each sample. For this purpose we need to extend the instruction mapping information that the compiler keeps for each method: Basically, we need to store a machine code mapping like a source-level debugger (from machine code addresses to Java bytecode in this case). This is already performed for methods that are compiled with the baseline compiler. For opt-compiled methods however the compiler only stores this information for the GC points. We extended the optimizing compiler so that it generates the bytecode index mapping for each machine code instruction, not only for GC points. From that point on we are able to count events for each IR (intermediate representation) instruction. Those event counts are updated by the sample collector thread periodically. This step is required to keep the IR data structures in memory after compilation. We found that the additional space overhead does not affect application performance significantly.

5. Approach

In this section we show an example optimization for data locality that applies the gathered performance data in a modified generational garbage collector [30].

5.1 HPM-guided co-allocation in the GC

Our system uses a generational mark-and-sweep garbage collector. It does bump-pointer allocation for young objects and copies matured objects into a mark-and-sweep collected heap. Tenured objects are managed using a free-list allocator that allocates objects into 40 different size classes up to 4 KBytes (=VM default setting) to minimize heap fragmentation. Larger objects are handled in a separate portion of the heap.

This collector offers better space efficiency than a pure copying GC (no copy reserve needed). On the other hand a copying GC is known to generally enhance data locality [9]. The goal of our co-allocation is to combine those two advantages, i.e. having a space-efficient GC that provides good data locality automatically by using feedback from the hardware: The online optimization consists of three parts:

1. Filtering of instructions of interest at method compilation time
2. Monitoring cache misses for individual classes and references
3. Nursery tracing algorithm that support co-allocation

The first part is performed for each method compiled by the opt-compiler. As a consequence the monitoring system does not consider instructions in non-optimized methods. However, this is not a major limitation since those methods are rarely executed (otherwise they would be selected for re-compilation by the JIT). Part two is done concurrently to the execution of the application. The sample collector thread periodically invokes the monitoring module that performs the bookkeeping and translates the raw data. The third part is implemented in the garbage collector where the cache miss data about field references are used to guide co-allocation.

```
class A {
  A x;
  A y;
  int i;
}
```

```
void foo() {

  ... = p.y.i;

}
```

```
I1: aload_2       // Local var p
I2: getfield   y; // Load field y
I3: getfield   i; // Load field i
```

Figure 1. Example bytecode for expression `p.y.i`.

5.2 Finding source instructions

For each method that is compiled with the opt-compiler (as selected by the AOS) the sample collector thread performs an additional pass to filter out instructions that must be monitored for cache misses in the HPM module: we are interested in reads/writes to objects that are referenced from another heap object. Initially, the compiler creates a mapping of instruction pairs: For each heap access instruction `S` it checks if the target address is loaded from a field variable `f` (also located on the heap). If yes, it saves a tuple (`S`, `f`). The motivation is that co-allocating the parent object with the child object increases the chance that both objects lie in the same cache line. This way the child object is implicitly prefetched when accessing the parent object. The opt-compiler computes this mapping by walking the use-def edges upwards from heap access instructions (field/array access, virtual calls and object-header access).

Figure 1 shows an example access path expression with its Java bytecode. Our analysis would create a mapping with instruction and field y (`I3`, `A::y`). For illustration we show the bytecode here - internally we actually use the actual high-level IR instructions that correspond to the bytecode. If we encounter a miss on `I3` (load of field `i`), we increase the event count for associated reference field (`A::y`). We keep a per-reference event count which tells the runtime system how many misses occurred when dereferencing the corresponding access path expressions.

5.3 Online monitoring

Samples from the HPM unit are buffered and processed in batches inside the VM: a sample is attributed to a reference field `f` if the source instruction `S` is among the instructions of interest (i.e. a mapping (`S`, `f`) exists). Currently we set the system up to monitor events in the application classes only and exclude events occurring inside VM code. This is not a limitation of the monitoring system itself, but just because the optimization deals with objects allocated in the user code.

The rate of events for each reference field is measured throughout the execution and this allows detecting phase changes in the execution or checking whether an optimization decision by the JIT or the GC had a positive or a negative impact. On many platforms, the effect of a data locality optimization is difficult to predict in general. A system that includes feedback based on a performance reporting unit allows an assessment of the effectiveness of an optimization step. If the transformation improved performance, the system can proceed normally. If the transformation reduced performance, either a different optimization step can be performed or it is possible to revert to the old code. This system is a step into

an performance-aware runtime environment that can judge which optimizations actually bring benefits and which do not.

5.4 Nursery tracing with co-allocation

When the GC hits an object that contains a reference fields during performing a nursery space collection it checks if it is possible to co-allocate the most frequently missed child object: we have to check if both objects together do not exceed the size limit for the free-list allocator. Object larger than this limit are allocated in a separate large object space. The VM keeps a list the reference fields for each class type sorted by number of associated cache misses.

When deciding to co-allocate two objects the GC just requests enough space to fit both objects. They will be assigned to the appropriate size class by the free-list allocator that manages the mature space. Without co-allocation the objects may - depending on their size - end up in different size classes. This would reduce spatial locality in the mature space.

Note that this approach may increase internal fragmentation because there is only a limited number of size classes (40 in our allocator) that do not cover each size exactly. The actual results depend on how co-allocated objects fit into their assigned size classes.

We chose the GenMS collector because we want to combine space-efficiency and good locality. None of the existing collectors provides this combination. Of course an optimized static copying strategy could achieve a similar benefit in many scenarios [25], but adapting to an individual application's memory access pattern proved to be important [17], and it has been shown that data locality optimizations often help in some cases and hurt in others. Detecting those cases at run-time is a strong argument for using performance counters for guidance.

6. Evaluation

6.1 Experimental platform

We carried out our experiments on a 3 GHz Pentium 4 with 1M L2 cache and 1 GB of main memory. The L1 cache for data is 16K. One cache line contains 128 bytes. The P4 has an out-of-order execution engine and can issue several instructions in parallel. It also includes hardware-based prefetching of data streams.

The system runs a Linux 2.6.16 kernel with the Perfmon2 patches and the corresponding libpfm library (version 3.2).

The baseline to which we compare our measurements is Jikes RVM 2.4.2 with the "FastAdaptiveGenMS" build configuration which is one of the most efficient configurations (default "production" build). It includes the adaptive optimizing JIT compiler [7] and a generational garbage collector with an Appel-style variable size nursery [6]. The mature space is managed by a mark-and-sweep collector. This collector is included with MMTk [10] - the garbage collection framework that comes together with the Jikes RVM.

We use a pseudo-adaptive configuration for the Jikes JIT compiler. Each program runs with a pre-generated compilation plan. This ensures that the compiler optimizes exactly the same methods and the variations due to the adaptive optimization system are minimized.

The different benchmarks are listed in Table 1. All timing results are averages over 3 program executions using the largest input size for the SPECjvm98 programs and the "default" input size for the DaCapo programs[3]. We also report the standard deviations for the execution times but found those to be very small in practice.

[3] The programs *chart*, *eclipse* and *xalan* were excluded because they are not compatible with version 2.4.2 of Jikes RVM.

db mtrt compress jess javac mpegaudio jack	Programs from the SPEC JVM98 benchmarks [29] with the largest workload (s=100) repeated 3 times.
antlr bloat fop hsqldb jython luindex lusearch pmd	Programs from the DaCapo benchmark suite (version 10-2006 MR-2) [11].
pseudojbb	This is a version of SPEC JBB2000 [28] with a fixed number of transactions (n=100000, max 6 warehouses).

Table 1. Benchmark programs.

The chosen sampling intervals are randomized by changing the lower order bits randomly (8 bits in our configuration). This should prevent us from measuring biased results by sampling at the same locations over and over.

6.2 Time and space overhead of runtime monitoring

In this section we show how expensive the runtime monitoring infrastructure is in terms of execution time and space overhead. Both must be reasonable to make optimization using runtime monitoring possible.

The systems needs to allocate additional memory for gathering detailed source-level performance data. First, there are buffers for temporary storage of the samples collected. The user-space library keeps an 80K byte buffer and the VM data collection thread stores the raw data in an `int[]` array of the same size.

In the VM we need additional tables to resolve raw samples to Java methods and bytecode. The space overhead of the additional meta-data in the VM is shown in Table 2. The second column (machine code) shows the size of the machine code generated by the compiler in KBytes. Column 4 (MC maps) shows the size of the machine code maps that are needed to resolve raw samples. For comparison, we show the size of the GC maps alone in Column 3. The last row shows the total size and the map sizes of the Jikes boot image. The boot image maps are pre-generated at compile-time and do not contribute to execution time. We can see that the machine code maps are 4 to 5 times as large as the GC maps, but the total sizes of the maps for an application are tiny compared to the maps that are contained in the boot image.

We consider only library and application classes and leave out VM internal classes at the moment because we do not consider them for optimization. Including these would just make the boot image larger but would not influence application performance. Currently, the whole boot image is about 9MB bigger than the original (increase of 20% from 45M to 54M). The maps for application classes take up to 5x the space needed for the GC maps. However, in absolute numbers the size of the maps generated is moderate (up to 1870K bytes for jython). Overall the amount is small compared to the maximal heap size

There is potential for improving the space efficiency of the machine code mapping to reduce the size of the boot image. We reused the existing implementation for GC maps and it would be possible to custom-tailor the data structure for our needs. But the

program	machine code	GC maps only	MC maps
compress	12	6	28
jess	20	12	43
db	7	4	20
javac	55	30	140
mpegaudio	71	31	168
mtrt	46	26	120
jack	40	22	111
pseudojbb	316	164	948
antlr	38	26	90
bloat	77	46	247
fop	8	4	16
hsqldb	117	67	290
jython	685	422	1870
luindex	119	58	316
lusearch	93	46	239
pmd	64	43	174
boot image	14975	10380	8260

Table 2. Space overhead: Size of machine code maps in KB.

runtime overhead of using the existing data structures is low enough to use it for our purpose.

Figure 2 shows the execution time compared to the original VM configuration without runtime event sampling using different sampling intervals from 25K to 100K. In this experiment we configured the system to monitor L1 cache misses. We measured the execution time for 3 different sampling intervals to evaluate the relation between sampling rate and execution time overhead. The "auto" configuration has a variable sampling interval because it adaptively changes the sampling interval at runtime. The reported numbers for execution time are averages over 3 executions of each program, and they include all overhead from mapping raw sample data.

For most programs the time overhead is proportional to the sampling rate (e.g. db and pseudojbb). A smaller sampling interval means higher sampling frequency and thus more data to be processed by the monitoring module. For others (e.g., mpegaudio) the constant portion of the overhead dominates. The absolute number of samples is not very high in these cases. The worst case is an increase of almost 3% for mtrt, compress and hsqldb with the smallest interval(25K). For the "auto" interval setting and an interval of 100K the average overhead is below 1% – a value that is low compared to software-only profiling techniques. Our experiments with data locality optimizations indicate that for our set of programs the largest interval is often sufficient to obtain enough coverage.

6.3 Effect of object co-allocation

Now we study the effect of the GC optimization that we performed using the runtime performance data. We compare the baseline with our co-allocating GC in different configurations and use L1 cache misses to guide the optimization. For the execution time we used a fully autonomous mode that adapts the sampling interval to obtain a certain number of samples per second. With a sampling approach the choice of an appropriate sampling interval is critical. The interval should be fine enough to give a statistically representative picture of the program behavior. But, since we are performing the sampling during program execution the overhead should also be reasonably low. The runtime overhead is proportional to the number of samples collected by the VM.[4]

[4] In automatic mode the only monitoring parameter is "samples/sec" - in practice we found that a default of 200 samples/sec provides reasonable accuracy and low overhead for all benchmarks programs.

Figure 3 shows the number of co-allocated object for different sampling intervals using a logarithmic scale. There are 2 programs (compress and mpegaudio) where no objects are co-allocated. They allocate mostly large objects which are placed in the separate large-object space by the allocator or only allocate few objects. Therefore, they have no candidate objects for co-allocation. The programs with a large number of co-allocated objects (db, pseudojbb, hsqldb, luindex and pmd) are less sensitive to the choice of the sampling interval: The largest interval is enough to cover most objects. In the remaining programs the number of co-allocated objects is several orders of magnitude lower, and co-allocation is more sensitive to the choice of the sampling interval. On the other hand, the impact on L1 cache performance is also less significant because the absolute number of objects is much smaller.

Performance impact

Figure 4 shows the number of L1 cache misses with co-allocation in the GC turned on relative to the baseline using a large heap size. In Figure 5 we summarize the impact on application performance using a range of heap sizes (1-4x minimum heap size).

For large heaps, there is a noticeable reduction in L1 cache misses using HPM-guided co-allocation for several programs (jess, db, pseudojbb, bloat and pmd). mpegaudio shows varying numbers (from -6% to +5%) that are not due to co-allocation (no candidate objects), but rather show influences from the event monitoring and processing. There is little or no effect on the other programs. From all benchmarks db gets the most benefit: 28% fewer L1 cache misses. This benefit translates into an execution time reduction of up to 13.9%.

Figure 6 analyzes the performance of db in more detail. Now we compare a generational copying (GenCopy) collector versus the generational mark-and-sweep (GenMS) with object co-allocation. The GenCopy collector generally improves spatial locality in the mature space over a non-moving collector - on the other hand it has a larger GC cost at small heap sizes [9]. This result is confirmed in our experiment. The maximum speedup versus GenCopy is only 10% vs. 13.9% compared to the baseline GenMS collector without co-allocation. We also see that GenMS + co-allocation outperforms GenCopy throughout all heap sizes (from 7% for large heaps to 10% for a small heap). This indicates that the GenMS collector with object co-allocation combines good locality with space-efficiency.

The reduction on L1 misses for jbb, one of the most memory-intensive programs from this suite, is only between 2 and 6%. The resulting speedup is up to 2% for large heaps. Here we observe that there are many frequently missed objects (2.4 million objects were co-allocated) and that the majority of those objects are relatively large (long[] arrays with a size of $>$128 bytes). As a consequence, optimizing for reduced cache misses at the cache-line level does not yield a significant benefit for this program. (Using TLB misses as driver for the optimization decisions does not improve the results.)

For the majority of the JVM98 benchmarks the number of co-allocated objects is rather small (in the order of thousands). There are not many mature objects that cause frequent cache misses: These programs have relatively small working sets and/or many young objects that do not benefit from better spatial locality in the mature space. Overall, three programs (db, pseudojbb, bloat) show a speedup, and 7 programs are slightly slowed down by using dynamic co-allocation. The worst case for large heaps is javac with -2.1% which is similar to the sampling overhead reported in Figure 2. Note that monitoring is turned on throughout the whole execution even when no candidate objects are found. The overhead could be reduced by turning off monitoring for most of the time in such a scenario.

For small heaps sizes the picture looks different. db is the only program that still shows a speedup at minimum heap size (9.3%).

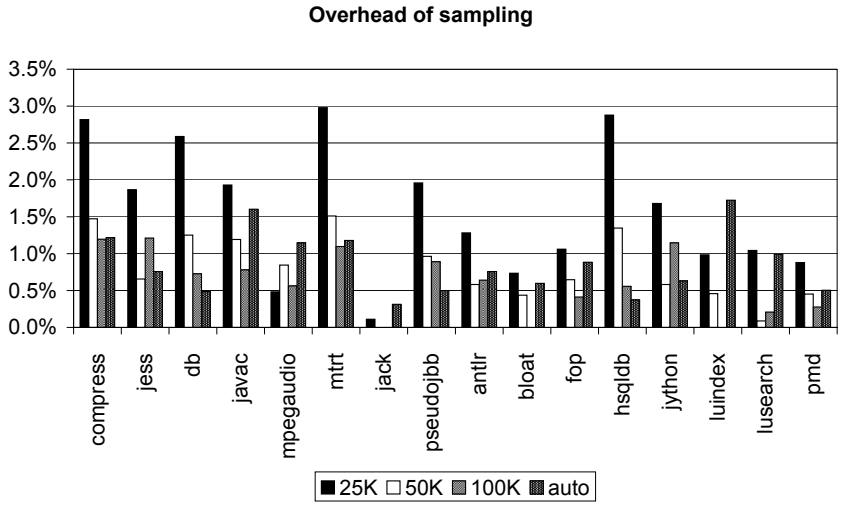


Figure 2. Execution time overhead compared to the baseline configuration with different sampling intervals (heap size = 4x minimum heap size).

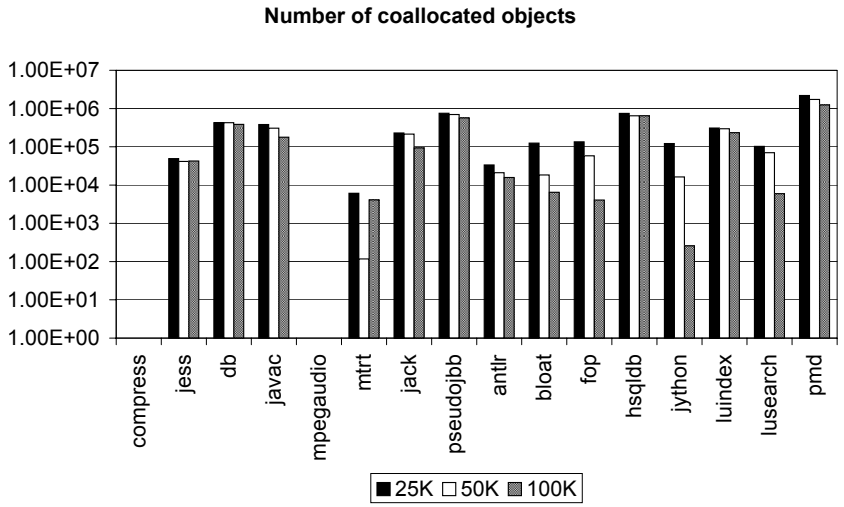


Figure 3. Number of co-allocated objects at different sampling intervals (heap size = 4x min heap size).

Figure 6. GenCopy vs GenMS with co-allocation

Generally, co-allocation yields better results at larger heaps. The larger the allocated chunks the more internal fragmentation exists due to the limited number of size classes in the free-list allocator. When running at the minimum heap size this space overhead factor gets more dominant and almost all programs are either slowed down or have a smaller speedup (e.g., db) with co-allocation.

6.4 Runtime feedback

To actually guide optimization automatically a VM needs accurate feedback. Figure 7 depicts two types of data that we collect for programs, here shown for the db benchmark: Figure 7(a) shows the cumulative total count of L1 cache misses when dereferencing the field `String::value`. The sharp bend for "dyncoalloc" occurs exactly when the co-allocation is switched on. The stepwise-constant shape of the measurement is caused by our batch-processing of samples in the monitoring module.

Figure 7(b) shows the L1 cache miss rate over time. It is locally quite volatile (in part also due to our monitoring infrastructure), but

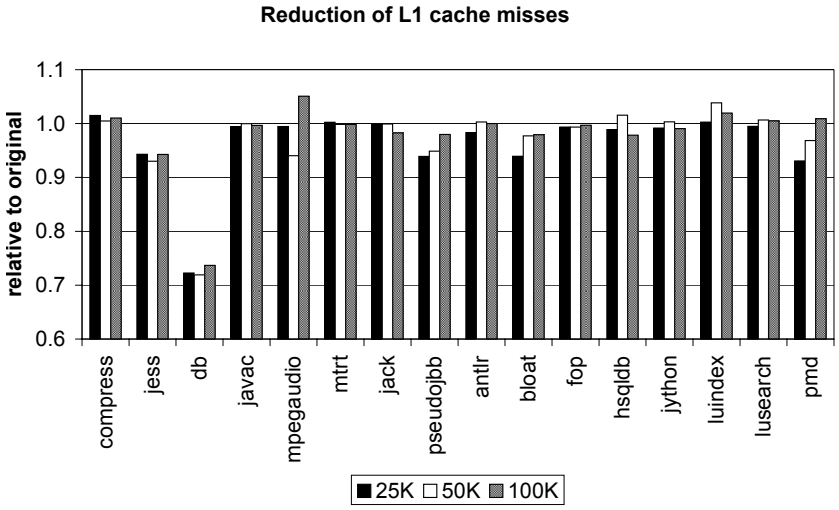


Figure 4. L1 miss reduction with co-allocated objects (heap size = 4x minimum heap size).

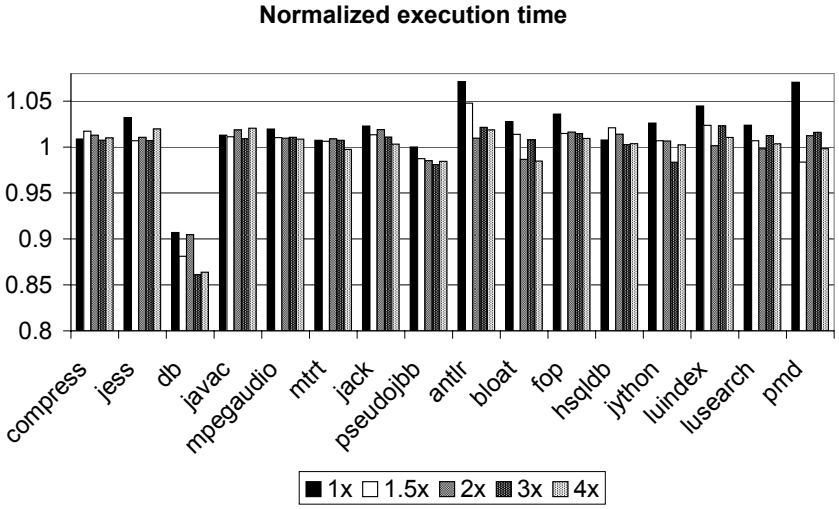


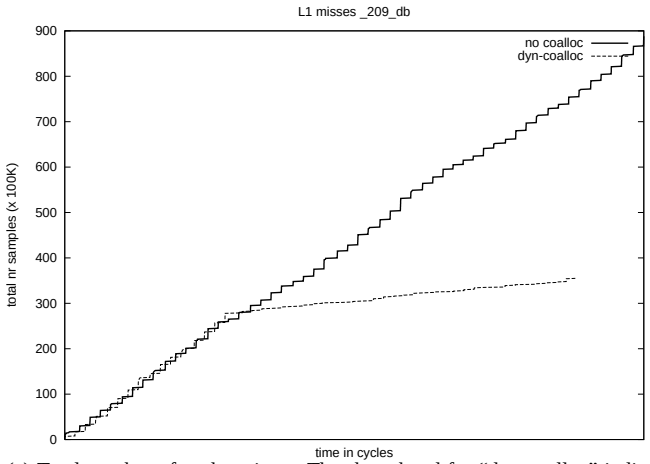
Figure 5. Execution time relative to the baseline for different heap sizes (sampling interval is auto-selected, heap size from 1-4x min heap size).

we can see the drop in the miss rate at the same time as in Figure 7(a) when co-allocation becomes active after the "warm-up" phase. The bold lines show the actual measured values. In addition we plot the moving average over the last 3 periods for both versions as thin lines. This metric follows the general trend without heavy local fluctuations. The precise association of the miss events with object types and references allows the VM to assess the effect of individual optimization decisions: in this case the internal `char[]` was co-allocated with the `String` object which resulted in a total reduction of misses on those objects by around 60%.

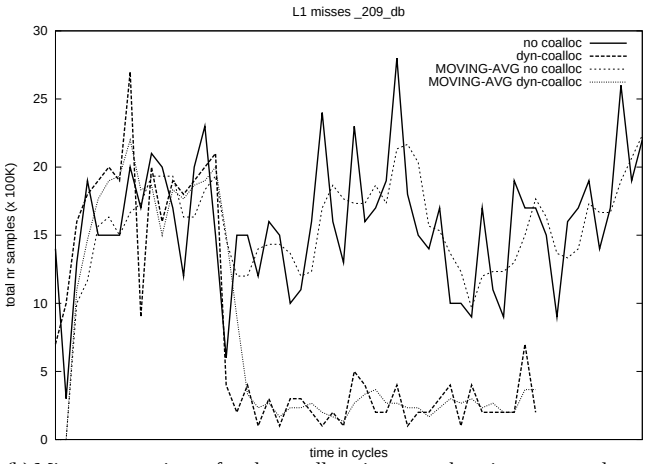
For long-running application the VM also needs to detect when an optimization has a negative effect on overall performance. To illustrate such a situation we show the cache misses over time when the GC happens to perform a poorly performing optimization in a controlled setting. Figure 8 shows the cache misses over time for `String` objects in `db` starting out with a good allocation order. We then instructed the GC manually to place one cache line of empty space (128 bytes) between the `String` and the `char[]` objects - effectively undoing the originally well performing setting. Monitoring the cache miss rate for individual classes allows the system to discover that this transformation does not improve performance, and after several measurement periods it triggers a switch back to the original configuration. Currently, a simple heuristic is used to determine when to switch, and we are still investigating suitable settings. Also, mature objects that are already co-allocated remain in place - only newly promoted objects will follow the new copying policy. Figure 8 shows the effect on the miss rate after switching back to the original allocation policy; the miss rate returns to its old value. We did not see such a situation where undoing co-allocation was necessary during our experiments with co-allocation – this may be more important for other optimizations.

7. Conclusions

We presented a system that uses the results of a modern hardware-based performance monitoring unit. As an example we discussed a



(a) Total number of cache misses: The sharp bend for "dyn-coalloc" indicates the time when co-allocation kicks in



(b) Miss rate over time: after the co-allocation starts the miss rate goes down

Figure 7. Effect of co-allocation: Cache misses sampled for `String` objects `db`

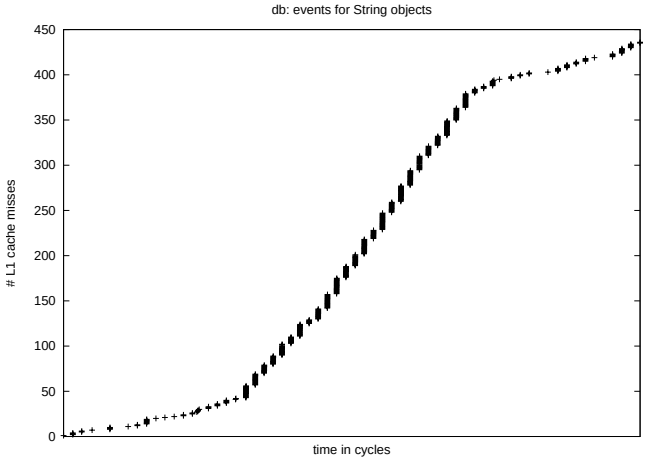


Figure 8. Cache misses sampled for `String` objects `db` with an poorly performing locality "optimization"

data locality optimization to show how such performance information can be used in a dynamic runtime environment. The overhead imposed by monitoring is reasonably low ($<$1% avg), and with appropriate compiler assistance it is possible to map performance-related events to source-level constructs. Our example optimization shows that a garbage collector with knowlegde about frequently missed objects and references can improve data locality and can detect at run-time if a data-locality optimization has a positive or a negative impact on performance. With the co-allocation technique for matured objects that is discussed here, L1 cache misses are reduced by up to 28%. The resulting application speedup is up to 14%, though this optimization is effective only for some programs. A more refined model of the micro-architecture in the compiler may be able to better exploit the performance data.

The infrastructure is flexible to allow compiler and GC implementers to include such information into their system as an additional source of runtime feedback. In our system the VM can actually "observe" the effect of data locality optimization. This is especially important since modeling the behavior of complex modern hardware architecture is very hard, and it is often a challenge to predict the effect of an optimization prior to performing the transformation. Feedback from the lower layers of the execution platform can be valuable information to guide such optimizations.

8. Acknowledgments

We thank the referees for their helpful comments.

References

[1] Perfmon project. http://www.hpl.hp.com/research/linux/perfmon/.

[2] IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. 2005.

[3] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proc. of Conf. on Programming Language Design and Implementation (PLDI 2004)*, pages 267–276, New York, NY, USA, 2004. ACM Press.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeno in Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPLSA 1999)*, pages 314–324, 1999.

[5] B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[6] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, 1989.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 47–65, New York, 2000. ACM Press.

[8] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 111–129, New York, USA, 2002. ACM Press.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS*

2004/PERFORMANCE 2004: Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems, pages 25–36, New York, NY, USA, 2004. ACM Press.

[10] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in Java with mmtk. In *Proc. of the Intl. Conf. on Software Engineering (ICSE '04)*, pages 137–146. IEEE Computer Society, 2004.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the Conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006)*, New York, Oct. 2006. ACM Press.

[12] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, Dec 1991.

[13] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proc. of the ACM SIGPLAN'99 Conf. on Programming Language Design and Implementation (PLDI 1999)*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[14] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI 2000)*, pages 13–26, New York, NY, USA, 2000. ACM Press.

[15] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 270–287, New York, NY, USA, 2004. ACM Press.

[16] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *Proc. of Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[17] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[18] X. Huang, B. T. Lewis, and K. S. McKinley. Dynamic code management: Improving whole program code locality in managed runtimes. In *VEE '06: Proc. of the Intl. Conf. on Virtual Execution Environments*, pages 133–143, New York, USA, 2006. ACM Press.

[19] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22(3):490–505, 2000.

[20] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proc. Conf. on Programming Language Design and Implementation (PLDI 2006)*, pages 239–251, New York, USA, 2006. ACM Press.

[21] K. Pettis and R. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN'90 Conf. on Prog. Language Design and Implementation*, pages 16–27, White Plains, N.Y., June 1990. ACM.

[22] S. Rubin, R. Bodik, and T. Chilimbi. An efficient Profile-Analysis framework for data-layout optimizations. In *Proc. of the Symp. on Principles Of Programming Languages (POPL 2002)*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[23] F. Schneider and T. Gross. Using platform-specific performance counters for dynamic compilation. In *Proc. of the Intl. Workshop on Compilers for Parallel Computing (LCPC 2005)*, Oct. 2005.

[24] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 13–25, New York, 2002. ACM Press.

[25] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *Proceedings of the 2006 Intl. Symposium on Memory Management (ISMM 2006)*, pages 52–63, New York, USA, 2006. ACM Press.

[26] B. Sprunt. Pentium 4 performance monitoring features. In *IEEE Micro*, pages 72–82, July–August 2002.

[27] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proc. of the ACM Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPLSA 2001)*, pages 180–195, New York, NY, USA, 2001. ACM Press.

[28] The Standard Performance Evaluation Corporation. SPEC JBB2000 Benchmark. http://www.spec.org/jbb2000/.

[29] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[30] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proc. of the Software Engineering Symposium on Practical Software Development Environments (SDE 1)*, pages 157–167, New York, USA, 1984. ACM Press.