RC25150 (W1004-072) April 20, 2010
Computer Science

IBM Research Report

Power-Performance Implications of Software Runtime Bloat: A Case Study with the SPECpower_ssj2008 Benchmark

Suparna Bhattacharya
IBM Systems and Technology Group

Karthick Rajamani
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758
USA

Manish Gupta
IBM Research

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Power-Performance Implications of Software Runtime Bloat

A Case Study with the SPECpower_ssj2008 benchmark

Suparna Bhattacharya
IBM Systems and Technology Group
bsuparna@in.ibm.com

Karthick Rajamani
IBM Research
karthick@us.ibm.com

Manish Gupta
IBM Research
mgupta@us.ibm.com

Abstract

This paper presents a study of the impact on performance and power consumption of software bloat for Java applications. With the SPECpower_ssj2008 benchmark, we create three distinct levels of object allocation - Alloc More, Alloc Orig and Alloc Less - to represent degrees of optimization/tuning for the creation of temporary objects. We then examine their impact on performance and energy-efficiency across a range of hardware and software configurations using real-time power and performance/activity instrumentation.

On a IBM HS21 blade server we see that the smarter allocation strategy, Alloc Less, can obtain a 59% improvement in energy-efficiency at peak load over the default strategy, Alloc Orig. On a POWER7 system we examine the sensitivity of the efficiencies of the allocation strategies to levels of multi-threading, cache sizes and JVM tuning. Further, interaction with power management based on dynamic voltage and frequency scaling (DVFS) shows synergistic benefits. DVFS reduces the detrimental impact of software bloat and can increase the benefits from bloat reduction - with DVFS, Alloc Less gets to 56% of the power for Alloc More, whereas without DVFS, its power is at 95% of Alloc More's. Experiments on a Nehalem-based server further confirm the energy-efficiency impact of software bloat reduction and its benefits to reducing heap sizes.

We also present an analysis of why the JIT compiler is unable to perform the key optimization of object reuse that can be applied by even an inexperienced Java programmer and present preliminary ideas on a simple annotation based technique to eliminate this bloat. With a new microbenchmark for object reuse we show the power-performance tradeoffs of object reuse in isolation. Finally, our preliminary analysis with the DaCapo tradesoap and tradebean benchmarks show that the choice of fine grain messaging/transformations can have a significant power efficiency impact.

1. Introduction

The role of power-efficient software is gaining recognition as an under-explored opportunity [13, 23, 33] in energy optimization of server systems. Advances in this area have largely focused on power-awareness in resource management software [23], query-optimizers [33] and exploitation of power-efficient hardware (e.g. DVFS, solid state storage, heterogeneous cores) [13, 16]. A related consideration has involved optimizing resource requesters to avoid unecessarily holding on to resources in a way that interferes with hardware and system power management [23].

We believe that there is another promising angle of software power-performance optimization, which deserves similar attention: the improvement in resource efficiency of typical operations in large framework based applications.

Studies of software runtime bloat reveal a pervasive pattern of excessive memory, processing and IO overheads incurred in these applications. Our own experience with analysis of several case studies corroborates these findings, e.g., we have observed instances of 6 IO copies/transforms of an input document per request, 60-70% data structure overhead, 1-2 MB temporary objects generated per transaction.

The culture of objects, a sea of abstractions, the use of computers as communicators and just-in-case programming have been cited as key software development productivity trends that lead to such systemic inefficiencies [21]. The lack of high level insight and the scale of analysis needed to optimize across components currently make it difficult for optimizers to keep up with these trends. As a result, it is not unusual to observe consumption of a gigabyte of memory per hundred users in applications which need to scale to millions of users or to find hundreds of thousands of method calls and objects being created to service a single simple request [21].

Recent research that studies the effects of object churn (a typical symptom of bloat) on performance and scaling describes and characterizes an allocation wall [32] which can potentially limit the scalability of programs with high object churn.

However, while we expect reduction of bloat to have an impact on power-performance, we are not aware of any empirical studies that validate this intuition and provide deeper understanding of the relationship between bloat reduction and power-performance improvement.

This paper makes the following contributions:

- It presents the first experimental study, to our knowledge, of the power-performance implications of software bloat. We present results for SPECpower_ssj2008, a well-known server side Java benchmark (and the first commercial workload benchmark requiring power consumption and energy-efficiency reports across the full range of system loads), for three modern server systems. We show significant energy efficiency benefits from reducing object churn, ranging from 7% to 59%.
- It includes a detailed cross platform evaluation of these implications across a range of hardware and software configurations covering different processors, multiple levels of multithreading, cache sizes, power management, JVM tuning and heap sizes.
  i. We find that workload tuning and system characteristics such as SMT and cache size both are important factors in the impact of bloat reduction, e.g. non-tuned (out of the box) JVM configurations show more than 20% improvement with lower bloat.
  ii. Using real-time component level power and performance activity instrumentation we then determine that the first order effects of runtime bloat in the form of temporary objects on energy efficiency occur through its impact on the memory hierarchy usage - processor cache effectiveness, off-chip memory latency/bandwidth and memory reference activity.
  iii. Next, we show that dynamic voltage and frequency scaling (DVFS) based power management has a synergistic relation with the benefits from bloat reduction, achieving 42% power savings between the highest bloat alternative and the original implementation at equivalent performance, compared to 3% savings without DVFS.
  iv. We also find that the better allocation strategy could tolerate significant heap size reduction with better performance. This has the potential to generate additional energy efficiencies from better DRAM low-power mode exploitation, higher degrees of workload/virtual machine consolidation and reduction in physical memory used.
- It shows how the access to higher level semantic insights can enable even non-experts in Java to exploit opportunities for significant bloat reduction that sophisticated runtime optimizers without such semantic knowledge are unable to accomplish. We suggest ways in which these kinds of insights could be generalized into annotations that can enable automated solutions for a related class of bloat patterns.

The rest of this paper is organized as follows. Section 2 presents background on the problem of software bloat and qualitatively describes how it impacts consumption of resources and power. Section 3 describes the problem of object churn in SPECpower_ssj2008 and presents an optimization to reduce it. Section 4 presents experimental results. Section 5 describes why a modern JIT compiler is unable to optimize the excessive allocation of objects, and presents preliminary ideas on an annotation based approach to deal with the problem in a semi-automated manner. Section 6 presents results for additional benchmarks and Section 7 describes related work. Finally, Section 8 presents conclusions.

2. Software runtime bloat and energy efficiency

Software impacts power consumption primarily through its hardware resource utilization at runtime. The power efficiency characteristics of the hardware-firmware resources determine the actual power consumption for a given utilization profile.

Software encounters a variety of hardware resources on any given system - compute engines on the processor, on-chip memory, off-chip memory and disk storage being typical components. Every system is designed/configured with a particular balance of these resources. An imbalanced use of these resources, can cause a performance wall and under-utilization of other resources. For example high cache miss rates caused by profligate use of objects/method calls can cause under-utilization of the processor cores. If the system hardware-firmware are unable to react to the underutilization of components to lower their power consumption the situation not only causes poorer performance but also results in lower energy efficiency.

The exact utilization of physical resources is tightly linked to the virtual resource management by the runtime systems. In general, we term the excessive usage of the number or capacity of any virtual resource by the software as software runtime bloat.

2.1 Symptoms of software runtime bloat

Large framework intensive applications exhibit symptoms of pervasive run-time bloat [21]: patterns of excessive activity or memory usage that typically span methods across multiple layers [9, 31], e.g.:

- the problem of object churn (high volumes of temporary objects [9], i.e. short lived objects that are allocated on the Java heap) and nested chains of data copies [31]
- long sequences of expensive transformations with nested transformations, including additional transformations for reusing existing parsers, serializers, formatters, [19, 21]
- high data structure representation overhead (typically 50%-80% [18], plus data duplication across layers.
- inflation of memory, protocol costs and method indirections relative to actual data and logic, due to dynamic type description and dispatch mechanisms [21].

The origin of this kind of systemic bloat is linked to the same software development trends that have undeniably been extremely successful in fueling the growth and widespread impact of software. Unfortunately, these overheads incur a run-time cost despite the best efforts of current Java optimizers, which lack higher level insight and are limited in their ability to perform optimizations that span components [9, 24, 31].

2.2 Impact of runtime data bloat on resource utilization

Bloat in long lived objects, due to high data structure representation overheads occupy JVM heap space, directly accounting for excess memory usage and indirectly contributing to processing costs like extra pointer dereferences, cache usage inefficiencies and data transfer sizes when subsequently copied.

Bloat manifested in short lived objects (i.e. unnecessary temporary objects causing high object churn) affect execution time and processing costs in terms of cache footprint, memory bandwidth usage (and even pathlength in case of costly initialization sequences). Temporary objects also cause higher memory pressure by spreading the memory footprint of the application because the memory can only be reused only after the next garbage collection cycle. This leads to larger heap sizes and/or excessive garbage collection. Additionally even when neither memory bandwidth or heap size is a constraint, these temporaries result in an increased memory activity because of inefficient cache utilization.

Both aspects of bloat lead to reduced power efficiency.

2.3 Impact of runtime data bloat on power consumption

Runtime data bloat increases energy consumption by increasing the memory capacity and bandwidth resources required per transaction or compute task. On energy proportional hardware the increase in resource consumption translates to proportional increase in energy; in other cases the increment may be lower but tagged on to a high base cost. Secondly, if the bloated usage affects a bottlenecked resource, then it can aggravate power-performance inefficiencies. Energy proportional hardware may consume less power in this situation than non energy proportional hardware because of the reduced utilization levels of the non-bottlenecked resources caused by poor scaling. Thirdly, though we do not explore this aspect in our paper, software bloat can cause indirect energy losses by raising startup/idle footprint and response times, hurting efficient power management.

Software bloat does not directly impact resource management, but efficient power-aware resource management can enable savings in resource utilization through the reduction of bloat to translate more effectively to system level power savings.

3. Reducing object churn in SPECpower_ssj2008

SPECpower_ssj2008 is a server energy-efficiency benchmark from SPEC [25] combining both power and performance measures for a workload at different load-levels. The workload is based on SPECjbb2005, but it has some significant modifications to inject transactions in bursts and to generate multiple load levels based on a maximum achievable transaction rate measured during initial calibration phases. The benchmark begins at 100% load and then at each subsequent step drops the intensity by 10%, finishing with a test at idle. During the entire run, the system power consumption is measured. The benchmark metric is an energy-efficiency score composed of a ratio of the sum of the transaction throughput for each load level to the sum of the average power for each load level (including an idle period).

SPECpower is the first commercial workload benchmark requiring power consumption and energy-efficiency reports as well as examination of these characteristics across the full range of system loads. Together with its implementation in a Java framework it offers a unique avenue for our studies on power-performance impact of runtime software bloat in the context of server-side Java.

Resource utilization characterization: According to prior characterization studies [12], the long lived data footprint of SPECpower_ssj2008 is approximately 50MB per warehouse, and about 8KB of temporary objects are generated per ssjop.

3.1 SPECpower_ssj2008 memory bloat analysis

SPECpower_ssj2008 is a fairly simple application compared to typical complex deeply layered enterprise Java workloads where software bloat tends to be most pronounced. For example, stack depths are small, more than 3 times lower than those observed in application server stacks.

3.1.1 Long lived data bloat

We collected heap dumps for a sample run of the benchmark on a 64bit JVM and analysed data structure health metrics using an offline tool [20]. Figure 1 shows the results.

While there was a 60% data structure overhead in long lived objects, the actual volume of the overhead is limited because the long lived data footprint is just 50MB per hardware thread (the benchmark uses 1 warehouse per hardware thread).

3.1.2 Temporary objects bloat or object churn

The volume of short lived objects, on the other hand, is large when the benchmark runs on system configurations that can achieve high throughput. For example, at 1 million ssj operations per second, 8 KB of temporary objects per transaction amounts to 8 GB/s of object churn.

If we could create a big enough variation in this object churn, we expect to observe perceptible power-performance

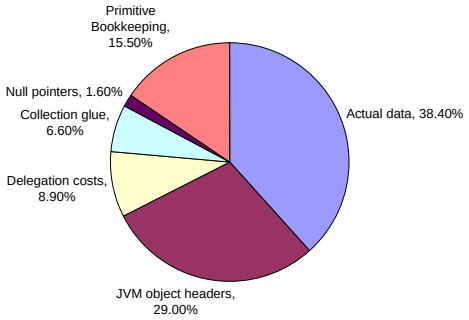

Figure 1. SPECpower_ssj2008 Data Structure Health

effects. Hence we decided to focus on identifying ways to vary the degree of bloat in uneccessary temporary objects, so that we could study power-performance implications of this category of bloat in SPECpower_ssj2008.

Turning off escape analysis in the JVM only changed the temporary allocations per transaction by 10%, and was not very effective in creating the substantial variation we desired. However, there is a particular allocation site that is responsible for about half of the temporaries. A review of the corresponding source code revealed an opportunity for significantly reducing these allocations using object reuse/pooling. We used this observation as starting point for our studies on the effects of reducing temporary objects bloat.

We also noticed another code fragment explicitly optimized for object reuse. Without this conscious optimization by the programmer, the object churn of the benchmark would have been higher. Unoptimizing this code provided a way to study the effects of increasing bloat in a way that can occur naturally in such applications.

3.2 Changes for reducing object churn (AllocLess)

Here we look at the particular allocation site in SPECpower_ssj2008 (Figure 2) for opportunity to significantly reduce the number of temporary objects generated with object reuse/pooling. This allocation site generates temporary String objects:

The routine `getLine()` is invoked (Figure 3) in a loop for copying the contents of the screen buffer into strings which are used to populate an XML transaction log (`populateXML()`), using the routine `putLine()`.

These strings get unreferenced when the transaction log is next cleared by invoking the routine `clear()` (Figure 4) so that it can be re-populated by the following transaction

```
class TransactionLogBuffer {

...

public String getLine(int line) {
     ---->    return new String(screenBuf[line]);
    }
..
}
```

Figure 2. Temporary string objects allocation site: getLine()

Instead of clearing the transaction log at the end of the transaction, the calling routines (Figure 5) invoke `clear()` during a subsequent transaction sometime before the next invokation of `populate()` of the same XML transaction log.

There are 80 lines of 24 characters each, which amounts to about 4-5KB of object churn per transaction. Based on the observation that the effective lifetime of these strings holds till the next transaction clears the log, this churn may be eliminated by reusing the memory of these strings when populating the XML log for the next transaction. We implemented this by maintaining a pool (a simple array) of string lines and reusing these string lines instead of issuing fresh string allocations in getLine(). Since String objects are immutable in Java, we had to use reflection to modify the contents of the underlying array in order to enable it to be reused. There are no synchronization overheads for the pool. The transaction log's object instance is only accessed by the thread that is running the corresponding warehouse's transactions.

3.3 Changes for increasing object churn (AllocMore)

Here we examine a code site where explicit object reuse optimization has been utilized in the original code. We examine how object reuse here can be disabled to model a situation without explicit optimization which would lead to increase in temporary object bloat.

The XMLTransactionLog maintains a line cache to reuse cleared document element lines (a LIFO pool) instead of creating them afresh for every transaction. The `putLine()` routine (Figure 6) is designed to first check for available entries in the line cache pool and reuse them, while the `clear()` routine (Figure 7) places cleared entries back into the pool.

Therefore for experimenting with the effects of increasing object churn, we disable the addition of cleared entries to the lineCache. This causes fresh allocations to occur at every transaction instead of reusing lines from the cache.

```
public class XMLTransactionLog {
   ...

   public void populateXML(TransactionLogBuffer log_buffer) {
      for (int i = 0; i < log_buffer.getLineCount(); i++) {
          putLine(log_buffer.getLine(i), i);
      }
   }
   ...

    private final void putLine(String s, int n) {
        ...
  --->  lineNode.getLastChild().getLastChild().setNodeValue(s);
        ...
    }
    ...
```

Figure 3. Populating XML log: Code that references the strings

```
public class XMLTransactionLog {
   ...
    public void clear() {
        ...
        while ((next_node =
          current_node.getPreviousSibling()) != null) {
            ...
       ---> lineNode.getLastChild().getLastChild().setNodeValue("");
            ...
            current_node = next_node;
        }
    }
}
```

Figure 4. Clearing XML log: Code that releases the references

```
class NewOrderTransaction extends Transaction {
    ...
    public synchronized void processTransactionLog() {
        ...
  --->  xmlOrderLog.clear();
        setupOrderLog();
        ...
        // create XML representation
  --->  xmlOrderLog.populateXML(orderLog);
        ...
    }
    ...
}
```

Figure 5. A populated XML log is cleared at the next transaction

```
private final void putLine(String s, int n) {
...
        // Check and see if a line element is available
        // in the line cache
        ...
        if (cacheLength > 0) {
   --->     // fetch a line from the line cache
            Node lineNode = lineCache.remove(cacheLength - 1);
            ssjDocument.appendChild(lineNode);
    ...
        }
        else {
   --->     // Create a new line element and append it to the document
            Element lineNode = (Element) document.createElement("Line");
            ssjDocument.appendChild(lineNode);
            Element newData = (Element) document.createElement("LineData");
            lineNode.appendChild(newData);
            Node new_node = document.createTextNode(s);
            newData.appendChild(new_node);
        }
    }
```

Figure 6. Code that uses entries from the lineCache pool

```
    public void clear() {
        ...
        while ((next_node = current_node.getPreviousSibling()) != null) {
            Node lineNode = baseElement.removeChild(current_node);
            ...
                // set the removed line's LineData Text Value to ""
                lineNode.getLastChild().getLastChild().setNodeValue("");
                // add the removed line to the lineCache
      ---->     lineCache.add(lineNode);
            }
            current_node = next_node;
        }
    };
```

Figure 7. Code that adds cleared entries to the lineCache pool

4. Power-performance impact: Experiments and Results

This section summarizes our different experiments to study the power-performance effects of temporary objects bloat in SPECpower_ssj2008 with different systems and conditions.

Our first set of experiments are on a 2-socket IBM HS21 blade with modest amount of memory, representative of the popular blade server space. We also had power management disabled during our studies on this system, again quite representative of vast majority of production servers today. More detailed experiments and analysis on two newer, larger server systems are then discussed. The first set is on a POWER7 processor based IBM POWER 750 server with detailed sub-system level power-performance instrumentation where we examine power-performance implications of software bloat and its sensitivity to SMT modes, cache sizes, JVM tuning, and and power management. The second set is with a Intel Nehalem processor based IBM x3650 M2 server to confirm cross-platform nature of our findings on the POWER7 system as well as some additional findings related to JVM heap size reductions.

4.1 Experiments on a blade server

System description: The system is an IBM HS21 blade server with dual processor sockets with quad-core Intel(R) Xeon(R) E5450 processors, 8GB DDR2 memory. The operating system kernel was Linux 2.6.33-rc8, configured without DVFS (no cpufreq). We ran a 64-bit JVM of IBM J9 (build 2.4, JRE 1.6.0). SPECpower_ssj2008 was configured to run with 1 JVM per socket.

We ran the benchmark at 2 load levels, 100% and 50% and compared the effects of disabling escape analysis and our code modifications for reducing and increasing bloat. Power measurements were collected using IPMI commands to the service processor that employs on-board measurement circuitry for accurate power measurements. The volume of temporary objects generated was computed by post processing garbage collection logs[1] using the IBM GCMV tool. Table 1 presents our results normalized against a baseline run of the original benchmark code (with escape analysis enabled by default).

100% load	ssjops (% of baseline)	power (% of) baseline)	temp objs per txn KB	ssjops per watt (% of baseline)
Original	100	100	8	100
DisableEA	92	99.2	8.68	92.8
Alloc Less	165	104	3	159
Alloc More	59	97.5	11	60.5
50% load	ssjops (% of baseline)	power (% of) baseline)	temp objs per txn KB	ssjops per watt (% of baseline)
Original	100	100	8	100
DisableEA	91	99.8	8.68	91.2
Alloc Less	164	101.6	3	161.6
Alloc More	57.5	98.4	11	58.4

Table 1. Power-perf impact on Intel Harperton

The allocation wall effect [32] seems to be fairly pronounced on this system, with substantial improvements in performance (65%) seen with reduction of temporary objects even with low GC overheads. The additional power consumption for this performance gain was just 4%, leading to a high performance/watt improvement (58%). We also experimented with larger heap sizes, using more JVMs and different bindings - we observed similar relative improvements in performance from software bloat reduction for these too.

Disabling escape analysis only resulted in a small reduction in temporary objects for these runs. Note that *Alloc More* also includes extra initialization costs for temporaries and not solely overheads due to allocation.

Reduced heap size runs A lower temporary objects allocation rate can potentially also enable lower heap sizes to be used. Reduced heap size runs are interesting from a power-performance perspective, because any significant savings in memory footprint could enable (i) low-power mode usage for inactive memory ranks, (ii) the use of a smaller memory configuration of the system for cost and power savings, or (iii) allocation of spared memory to another virtual partition for improved workload consolidation with same system configuration/power envelope.

We compared the effects of a 4 fold reduction in heap size on the two alternatives (*Alloc More* and *Alloc Less*). The results are shown in table 2, normalized with respect to the original baseline run with full heap.

	heap size (relative)	ssjops (% of baseline)	power (% of) baseline)	% time in GC
Alloc More	100%	59	97.5	1.2
Alloc More	25%	48.6	97.8	18.6
Alloc Less	100%	165	104	0.7
Alloc Less	25%	130.5	102.8	13.6

Table 2. Reduced heap comparisons on Intel Harperton at 100% load level

We observe that a significant fraction of time is spent in GC - a few global collections and a large number of nursery collections. This leads to a substantial degradation in performance and energy-efficiency (for same memory configuration) for both variants of temporary object allocations. Even so, the alternative with less bloat, Alloc Less, has a 30.5% better performance than the original allocation even at one-fourth the original heap size.

Based on the promising results from these initial experiments, we decided to perform more detailed investigation of the effects on larger server systems with more memory capacity and bandwidth, and power management capabilities. We expect the allocation wall effect to be not as pronounced on such systems.

4.2 Experiments on an IBM POWER7 processor platform

The POWER7 processor is the latest in the IBM POWER processor family. It has 8 4-way multi-threaded cores for a total of 32 threads per procesor, separate 32-K L1 data and instruction caches per core, 256K L2 cache per core and a total of 32MB of shared on-chip L3 cache. Our experimental system is a prototype IBM POWER 750 server with 4 POWER7 sockets interconnected by an SMP fabric and 4x4GB DDR3-1066 dimms per socket, for a total of 64GB of memory. The system software stack consists of the POWER hypervisor running a pre-release AIX 6.1 level with 32-bit IBM J9 Java sdk (1.6.0).

POWER7 processor is designed for energy-efficiency and dynamic power management support [28] incorporating the second-generation of IBM EnergyScale [17] power management stack. The primary power management capability examined in our study is Dynamic Voltage and Frequency Scaling (DVFS) of the processor. Second-generation load-based DVFS scaling algorithms for IBM POWER platforms of those presented by Rajamani et al [22] are employed in some of the evaluations discussed below. Power management and monitoring are built-in facilities provided by real-time firmware running on a dedicated power management controller, called *TPMD*. Traces of fine-grain power measurements at the system and sub-system (processor socket and memory sub-systems) level, power management actions

[1] obtained by specifying the -verbosegc JVM parameter

like frequency changes and synchronized memory controller statistics are obtained from the TPMD for the results discussed in this work.

4.2.1 Impact of allocation strategies

We again compare the default allocation approach with the less and more bloat options of Alloc Less and Alloc More, on the POWER7 system. The measured temporary object space allocation per transaction were 6.9KB/txn for original, 1.8KB/txn for Alloc Less and 9.8KB/txn for Alloc More. Figure 8 shows the impact of this on the memory controller and memory power statistics.

Reducing software bloat reduces memory traffic to 67% resulting in a lowering of memory power consumption to 87%, the memory bandwidth per ssjop dropped to 63%. At the other end, Alloc More, has 134% of the Mem BW/ssjop as the original allocation. However, the impact of this increased Mem BW demand per transaction on the total bandwidth and memory power consumption is much less. The reason for this is that the overall throughput takes a significant hit for Alloc More because of increase cache pressure. While the memory controller's capacity is not taxed by the increase in bandwidth demands, the additional latency for traffic from on-chip cache to off-chip DRAM impacts the performance of Alloc More significantly.

This impact can be confirmed in Figure 9. The Norm score bars reflect normalized SPECpower_ssj2008 scores that are the energy-efficiency metric for the benchmark and Norm Perf and Norm Power bars the aggregate Performance (numerator) and Power (denominator) components for the efficiency score. So there is a 25% drop in performance for Alloc More while there is a 7% increase in performance for Alloc Less. With near identical power consumption, the performance improvements have very similar impact on the energy-efficiency score too.

4.2.2 Impact of Multi-threading

The experiments done earlier exploit the full 4-way multi-threading capability of the POWER7 processors. The processors also support lower levels of multi-threading. For our next set of experiments we examine the impact of software bloat with just two threads enabled per core. Figures 10 and 11 show the memory statistics and impact summary for the benchmark running in SMT2 mode comparing both Alloc More and Alloc Less with the original allocation approach, Alloc Orig. The memory statistics show that Alloc Less provides very similar improvements and Alloc More similar degradation in memory statistics for SMT2 mode as they did for SMT4 mode (captured in Figure 8), with slightly larger improvements in BW/ssjop metric for Alloc Less. When we look for the impact of these improvements, we see that Alloc More does suffer in a similar fashion in SMT2 mode too (79% of Alloc Orig's score in SMT2 mode and 76% in SMT4 mode). However, Alloc Less achieves

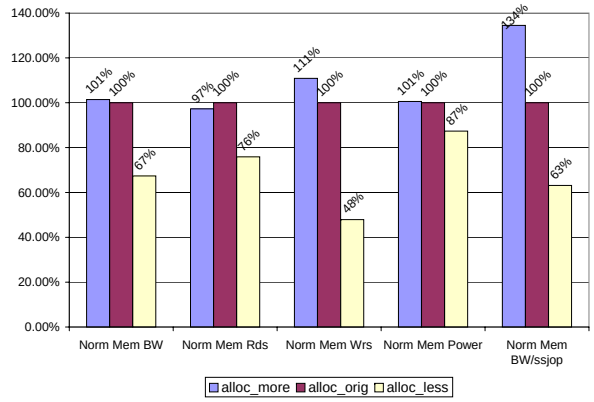

Figure 8. Memory Statistics for SPECpower_ssj2008 on Power7 system

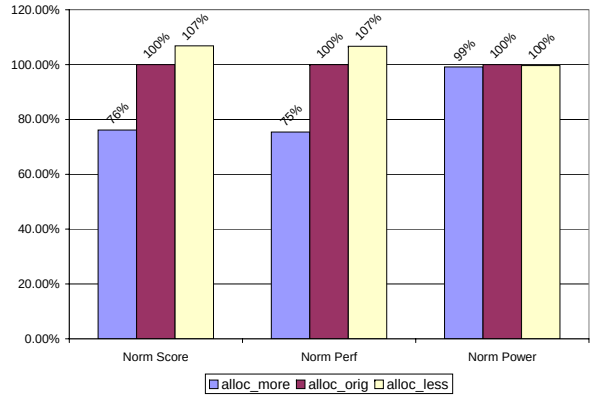

Figure 9. Impact Summary for SPECpower_ssj2008 on Power7 system

a much smaller improvement than it did for SMT4 mode - 102% in SMT2 mode versus 107% in SMT4 mode.

The main difference between the SMT modes is the cache pressure being roughly halved for SMT2 versus SMT4. Alloc More that worsens this cache pressure still produces a similar net degradation performance. However, with the cache pressure being relatively less with the original allocation, the improvement in memory usage that Alloc Less brings gets muted on overall performance.

All other results on the POWER7 system are for experiments conducted in the full 4-way SMT mode.

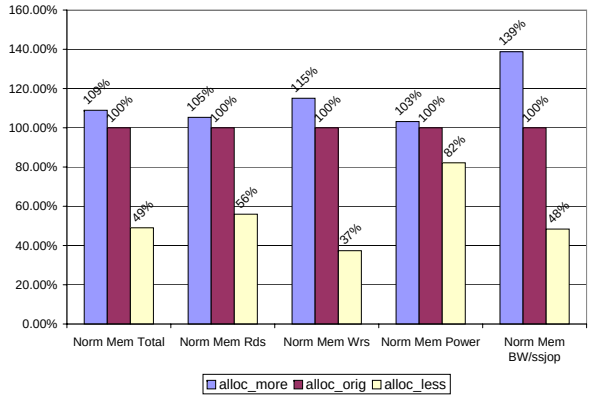

Figure 10. Memory Statistics for SPECpower_ssj2008 on Power7 system in SMT2 mode

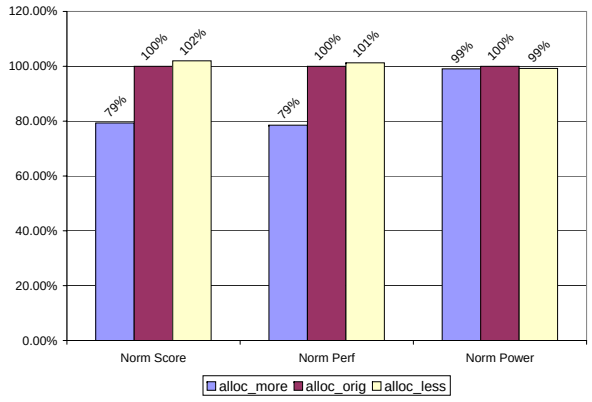

Figure 11. Impact Summary for SPECpower_ssj2008 on Power7 system in SMT2 mode

4.2.3 Cache size impact

With Alloc Less showing a higher performance impact in SMT4 vs SMT2 mode, possibly from additional cache pressure, we decided to examine if our relatively large on-chip cache was hiding some of the potential degradation effects of software bloat i.e. if reducing software bloat with Alloc Less could have a more pronounced impact if we had smaller processor caches. Configuring each of the processors with 16MB of L3 cache we compared the relative performance of Alloc Less with the original, Alloc Orig. The results are summarized in Figures 12 and 13.

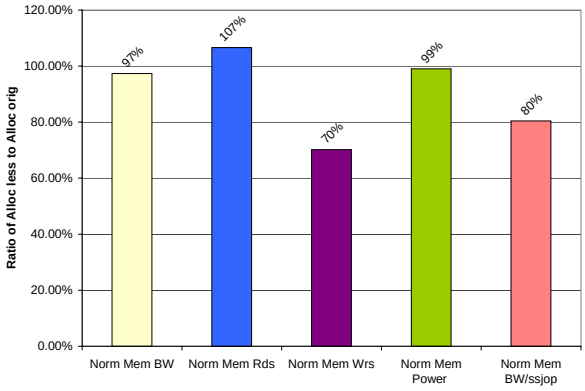

Figure 12. Memory Statistics for SPECpower_ssj2008 on Power7 system with Half Caches

With just half the cache size, we actually see a narrowing of the gap between Alloc Less and original in the memory statistics (e.g. Mem Bw/ssjop at 80% instead of 63% as seen in Figure 8). However, the net impact statistics in Figure 13 show the full picture. There is a 19% improvement for Alloc Less over Alloc Orig on the energy-efficiency score from a 21% increase in performance, as opposed to a 7% increase in performance with 32MB caches. From the memory statistics it is clear that this bigger relative performance improvement for Alloc Less is not from reducing memory traffic per ssjop but from the increased sensitivity to cache capacity and latency with the smaller caches. So it is clear that the large on-chip caches of the POWER7 play a key role in mitigating software bloat.

4.2.4 Impact of tuned workloads

For SPECpower_ssj2008, significant tuning efforts were undertaken to obtain good performance on the platform. While system vendors can invest tuning efforts for specific benchmarks, often system users may not be able to invest significant efforts for tuning their code. To study if our observations are unduly biased by working with a well-tuned settings for benchmark, we performed experiments with out-of-the-box run scripts for the benchmark i.e. only the default JVM and OS settings were used. For the non-tuned runs we measured the allocation rates at 8.23 KB/txn for original approach and 2.87 KB/txn for Alloc Less. Figures 14 and 15 provide the corresponding relative improvements for Alloc Less without benchmark tuning. In the memory statistics, while there appears to be similar reduction in Mem BW/ssjop the reduction in memory bandwidth and power are smaller. The reason again appears to be increased throughput for Alloc Less compared to the original, as confirmed by the summary impact statistics in Figure 15. The improvement

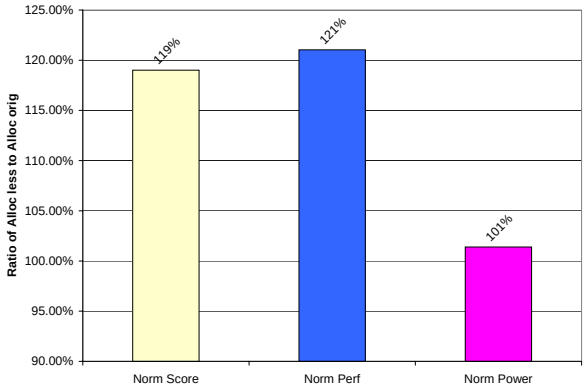

Figure 13. Impact Summary for SPECpower_ssj2008 on Power7 system with Half Caches

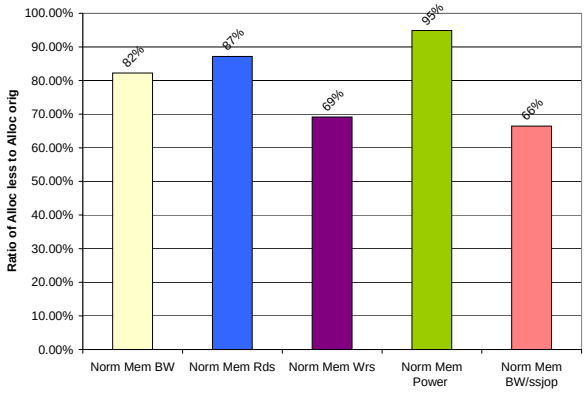

Figure 14. Memory Statistics for SPECpower_ssj2008 on Power7 system without tuning

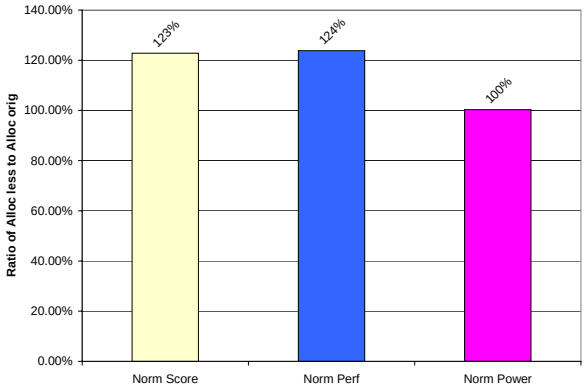

Figure 15. Impact Summary for SPECpower_ssj2008 on Power7 system without tuning

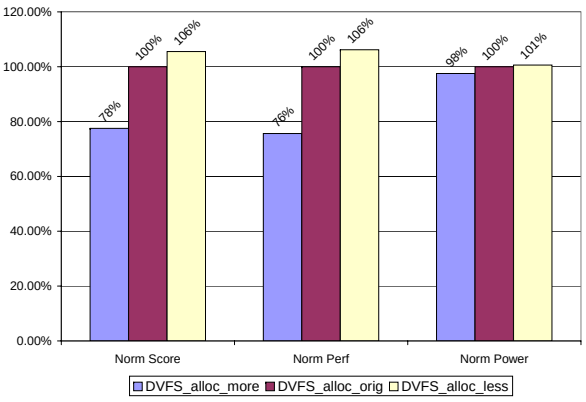

Figure 16. Impact Summary for SPECpower_ssj2008 on Power7 system with DVFS

in throughput is 24% for Alloc Less as opposed to 7% with the tuning (shown in Figure 9. There is a corresponding improvement in the energy-efficiency score with power being nearly the same again. Thus we can see here that non-tuned workloads might see an even bigger performance/efficiency impact from software bloat than tuned executions.

4.2.5 Impact of power management

Next we examine the influence of power management on the impact of software bloat by enabling the processor voltage and frequency scaling algorithms for energy reduction implemented in the TPMD. With full caches and tuning Figure 16 shows the impact for both the introduction of more software bloat with Alloc More and the reduction of bloat with Alloc Less. There appears to be a marginal reduction in the impact of software bloat with the relative score for Alloc More coming in at 78% (versus 76% in Figure 9 without DVFS) and the relative score for Alloc Less being 106% (versus 107% in Figure 9 without DVFS).

Continuing further, we conducted the half cache and no-tuning comparisons of Alloc Less with original but now with power management enabled - the results are captured in Figures 17 and 18. In both these scenarios, the processor is working at less than its full capacity even at peak load allowing the DVFS algorithms to take advantage of the exposed slack for power reduction. For the without tuning scenario,

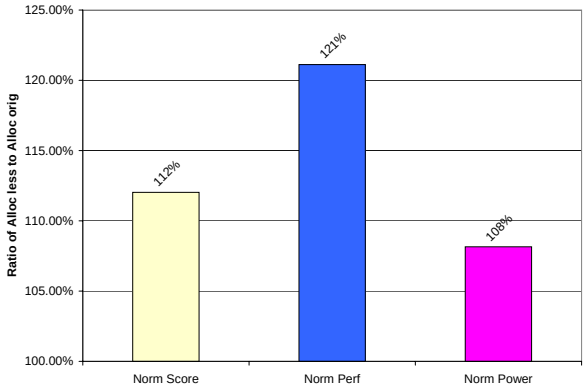

Figure 17. Impact Summary for SPECpower_ssj2008 on Power7 system with half cache and DVFS

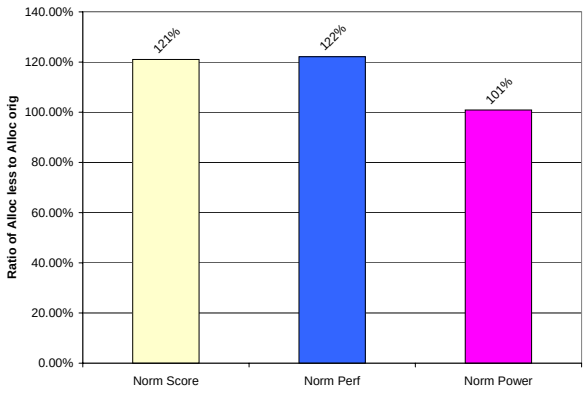

Figure 18. Impact Summary for SPECpower_ssj2008 on Power7 system without tuning and with DVFS

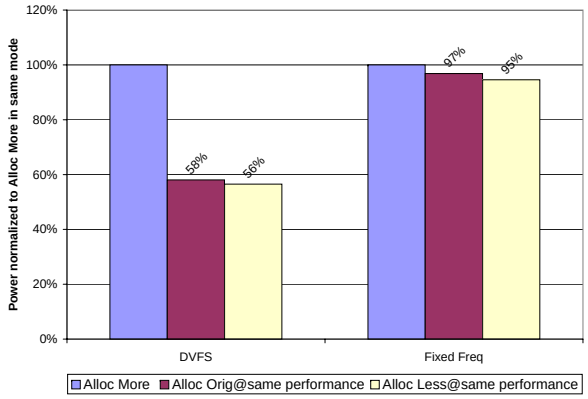

Figure 19. Impact of DVFS on power consumption with bloat reduction, at peak performance of Alloc More

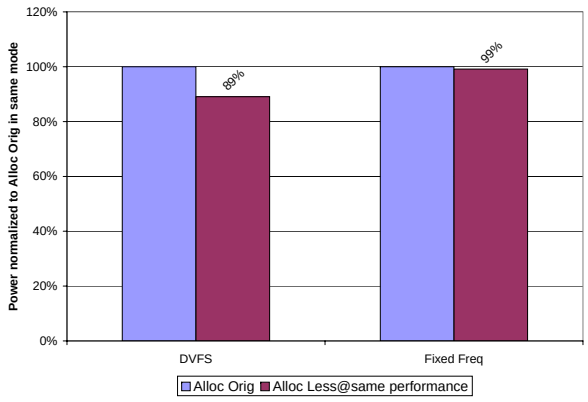

Figure 20. Impact of DVFS on power consumption with bloat reduction, at peak performance of Alloc Orig

the exposed slack is so big that the DVFS algorithm can meet the demand with the lowest frequency for both Alloc Orig and Alloc Less (even for its 22% greater performance). So the power numbers are same leading to the performance ratio being reflected as also the ratio for the energy-efficiency score. For the half cache scenario Alloc Less needs to use higher frequencies for meeting the additional demand that it is able to satisfy and so has a higher power cost (108%). Consequently this lowers the score improvement to 112% for a performance improvement of 121%.

A different view of the impact of reducing software bloat is presented in Figures 19 and 20. Here the allocation strategies are compared on their power consumption at identical performance levels. In Figure 19 the comparison point is at the peak performance (100% load level of SPECpower_ssj2008) of Alloc More. In Figure 20 it is at the peak performance of Alloc Orig (Alloc More cannot achieve that performance and so is not shown in this figure). From these figures it is clear that there is a synergistic benefit to reducing software bloat in the presence of DVFS - the power reductions from bloat reduction are visibly greater when DVFS is simultaneously being exploited for energy reduction (DVFS bars for Alloc Orig and Alloc Less are shorter than the Fixed Freq bars).

While the focus of this section of our study is the interaction between DVFS and software bloat, we also note

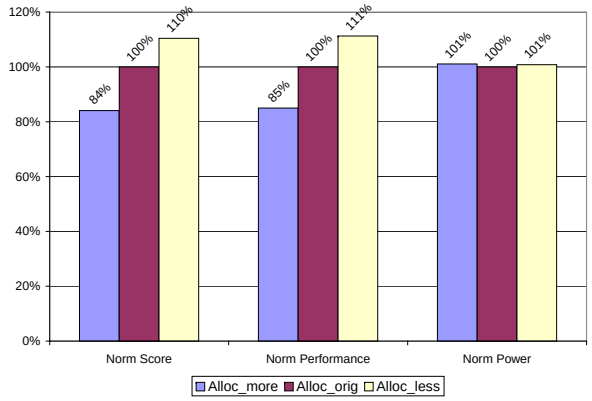

Figure 21. Impact Summary for SPECpower_ssj2008 on an IBM x3650 M2 server

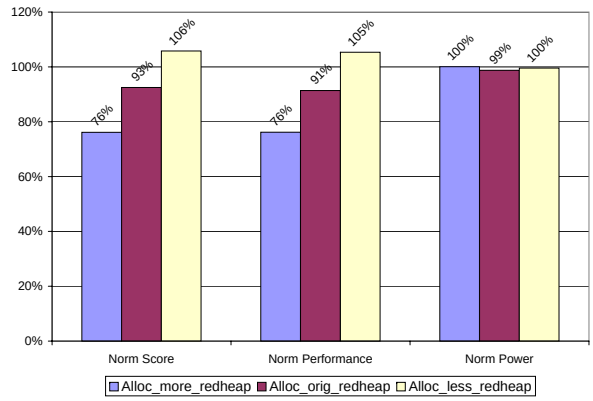

Figure 22. Impact Summary for SPECpower_ssj2008 on an IBM x3650 M2 server with reduced heap size

that usage of DVFS by itself brings significant improvement to the energy-efficiency score with no performance loss. For example with the original allocation approach we found 67% (fullcache with tuning), 77% (no tuning) and 89% (half cache) improvements in score for DVFS over the fixed frequency runs with the same performance.

4.3 Experiments on Intel Nehalem processor platform

The Nehalem processor used in our study is an Intel(R) Xeon(R) X5570 in a dual-socket IBM x3650 M2 system - each processor has 4 cores with two threads per core. Power measurements on this system were done at the system-level with periodic power samples obtained through IPMI communications with the on-board service processor. System was running Linux 2.6.33-rc8-mm1 with 64-bit IBM J9 JVM, Linux-based frequency scaling for energy savings was enabled.

Figure 21 shows the impact for the three temporary object allocation strategies. The results are quite similar to those on the POWER7 platform (comparable result there is captured in Figure 16). Software bloat has noticeable impact on the energy-efficiency score for the benchmark primarily through its influence on performance through cache pressure impact.

Figure 22 shows results for runs with a reduced heap size - total heap size was reduced in the ratio of the object allocation rates for Alloc Less and Alloc Orig (37.5%). The statistics are normalized to Alloc Orig's with full heap. As expected the performance worsens with a reduced heap size. The gap between the allocation strategies also widens in this more resource-constrained environment. It is encouraging to note that that with a smarter allocation strategy, like Alloc Less, performance is still better than the original (105%) even with significant heap size reduction. As noted before, heap size reduction has the potential to generate greater efficiencies from better DRAM low-power mode exploitation, enabling higher degrees of workload/virtual machine consolidation and reduction in physical memory used.

4.4 Summary

Following are the key observations from our experiments with SPECpower_ssj2008 on the three server platforms:

Runtime software bloat in the form of temporary object allocation has significant energy-efficiency impact, primarily through its impact on realizable peak performance. Secondary impacts include higher memory traffic and power consumption. The relative gains of addressing runtime software bloat is most dramatic in the smaller memory capacity blade server system with about 65% improvement in performance with better object allocation methods (Alloc Less) at peak load. Evaluations for the larger servers were done with full benchmark runs with the benchmark scores being 107% and 110% for the POWER 750 and x3650 M2 servers.

The first order impact is through its impact on the memory hierarchy usage - processor cache effectiveness and/or dependance on off-chip memory latency and bandwidth. Software bloat's impact is more acute when the cache resources are already strained with high multi-threading or with relatively lower cache capacity.

Poorly tuned JVM executions appear to suffer disproportionately from the issue of software bloat. Reducing bloat had bigger impact for out-of-the-box benchmark execution.

Power management in the form of processor DVFS has very complementary impact with addressing software bloat. On the one hand DVFS slightly reduced the energy-

efficiency degradation caused by software bloat, on the other it also significantly improves the energy-efficiency improvements brought about by reduction in bloat. At the equivalent performance level of Alloc More at 100% load level, the better allocation strategies of Alloc Orig and Alloc Less consume 58% and 56% lower energy, respectively, with DVFS and 97% and 95% lower energy, respectively, at a peak fixed frequency operation.

Suitably addressing runtime software bloat was found to signficantly aid heap size reductions. The better allocation strategy, Alloc Less, obtained improvements over the original allocation even with significantly reduced heap sizes (30.5% performance improvement for Alloc Less with a heap size of 25% for the HS21 blade server and a 6% improvement in energy-efficiency score with a heap size of 37% for the x3650 M2 server).

5. Discussion: Towards (semi) automated solutions

Our analysis of power-performance implications of reducing object churn has relevance for both manual code refactoring and automated solutions. Offline analysis techniques (combining static and dynamic analysis for example) have been proposed for aiding detection and refactoring of code to reduce object creation, [3, 9]. Dynamic runtime optimizations are most convenient but incur a power-performance cost, which affects evaluation of cost-benefit tradeoffs.

We were curious to understand what would it take to automate the kind of changes that we made to SPECpower to reduce object churn at runtime. A significant proportion of the churn was attributable to a specific allocation site which we were able to hand optimize with ease (even as non-experts). Therefore, we were initially surprised to find that sophisticated production JITs were not already exploiting this opportunity for optimization.

We created a testcase with a few simplified variations of the same code pattern to study the factors limiting the extent of optimization possible. One of the commonly cited reasons for limited effectiveness of runtime escape analysis for component based applications is the cost and scale of (interprocedural) analysis and challenge with inlining calls to sufficient levels required. We experimented with specifying options to the JIT for targeted inlining and hot method assumptions, but observed that the compiler still did not detect the opportunity even with several levels of simplification of the code example.

This could have been merely an artifact of the particular JVM implementation that we chose for our experimentation. However, a closer analysis reveals a few reasons why generalizing this could be non-trivial for runtime optimizers.

5.1 Potential Difficulties:

Globally escaping root scope The strings are referenced through several nested field levels off the transaction object on heap, which in turn is accessed via several levels of nesting (including a runnable class) from an anscestor class that is reachable from a different thread.

Cross transaction escape-capture The clear() routine which dereferences the strings is invoked inside the next transaction, hence the strings escape the transaction boundary. [The loop that iterates over transactions is the nearest common ancestor that spans the lifetime of the strings]

Field re-assignment does not ensure Bypass The clear() routine effects a dereference of the allocated strings by iterating over the line elements in the XML document object and resetting the value fields to "", but the analysis cannot perform a "kill" ("bypass" in [6]) as it cannot ensure that all previous references are killed.

5.2 Exploiting high level insight

Is there some generally applicable higher level semantic insight that could be passed to the runtime optimizer to enable it to exploit such opportunities more effectively ? This is an interesting question with relevance beyond narrow optimizations for this specific code snippet. Many cases of runtime bloat and their specific solutions are only apparent in the context of higher level knowledge and assumptions that are not explicitly available to the underlying optimizers.

In this particular case, we used the following high level insights for creating the variation with reduced bloat:

1. A transaction processing object instance executes one transaction at a time and is not concurrently accessed by any other thread, even though it may be reachable globally through the transaction managers list maintained for overall coordination. Hence no other thread can cause a reference to the strings to escape globally between a populate() and the subsequent clear(). Further no synchronization is required for the object pool of string lines.
2. A transaction processing object has two kinds of state, (i) state that is referred/reused/cached across transactions and (ii) state that is limited to the lifetime of a transaction. The latter kind of state can be captured at a subsequent invokation of the transaction processing method which corresponds to the next iteration/request using the same transaction object. Observing cross-transaction escape-capture relationships reveals this distinction.
3. There is an escape-capture or escape-reuse relationship for data referenced by the the populate() & clear() methods of the transaction log object. In that sense these are complementary methods: anything allocated in populate is reusable after a clear if not already being reused (e.g. the line cache is already reused, while the strings are captured and hence candidates for reuse).

6. Discussion: Other workloads

The detailed experiments with SpecPower_ssj2008 enabled us to isolate the effects of a single category of bloat using

well-understood natural control points in a non-trivial application context of an established server side Java power-performance benchmark.

The degree of impact of temporary objects reduction on performance is workload dependent, as has been established from results on object churn reduction in prior work [24, 32]. [32] defined a simple allocation microbenchmark, called "AllocMark" to study the costs of pure allocation. We extended this to create a microbenchmark called "AllocReuseMark" which incorporates object reuse for a percentage of the allocations. Power and allocation rate measurements were collected on the Intel Nehalem system running an instance of this benchmark on each processor/hardware thread for different levels of object reuse Table reftable:allocreuse. The reused bytes are always cache resident, hence 50% reuse doubles the effective performance. This improvement is associated with a increase in power consumption, but it is relatively smaller than the performance gain, resulting in a net gain in energy efficiency (similar to the DVFS results with reducing SpecPower bloat).

Reuse	KB/sec (% of base)	Power (% of) base)	KB/watt (% of base)
0%	100	100	100
25%	133	102.6	128
50%	199	105.3	190
75%	387	119	328

Table 3. AllocReuseMark comparisons on Intel Nehalem

It is important to note that excess temporaries is usually only a symptom of bloat. For example, it may be indicative of excess transformations and layering inefficiencies. Addressing the underlying causes rather than the symptom is therefore expected to result in much bigger gains in efficiency.

To gain an assessment of potential power-performance implications of such compounded patterns of bloat arising out of layering and transformation costs for a more complex server workload, we ran comparisons of the DaCapo [26] tradesoap and tradebeans benchmark. These application server benchmarks have been included in the recent release of DaCapo version 9.12 (December 23, 2009). These are based on the Apache DayTrader J2EE workload and are executed within the Geronimo application server utilizing the Derby in-memory database. Both tradebeans and tradesoap execute the same underlying workload, except that tradesoap uses indirect calls through SOAP from the client (like many real workloads do) while tradebeans executes direct calls on the server. Thus comparing these two configurations is useful for studying the costs of fine grained messaging, layering and transformation. Excesses of such costs are typical in patterns of software bloat observed in real applications.

Our preliminary observations indicated a 15% increase in power consumption, with close to 2X slowdown in performance in tradesoap vs tradebeans, resulting in an overall 2.27X degradation in power efficiency. We noticed that tradesoap generated almost 10x higher footprint of temporary objects and caused a significant increase in CPU utilization. This shows that fine grain messaging and associated transformations can have a significant power efficiency cost, especially for short high volume transactions and so abuse of it for scenarios where significant levels of dynamism are not needed should be avoided.

7. Related work

Analysis and measurement of software bloat Mitchell, Sevitsky and Srinivasan [19] define metrics based on modeling runtime information flow to classify and characterize the nature and volume of data transformations executed, but these measures have not been automated till date. The notion of data structure health signatures proposed by Mitchell and Sevitsky [18] has been used very effectively in characterization and automated measurement [20] of Java memory bloat in long lived heap objects. This is a relative measure of total memory bytes consumed by actual data vs associated representational memory overhead.

For some categories of bloat, including the problem of temporary objects bloat which we observe in our case study, where an explicit model may not be available for distinguishing overhead from necessary data or activity, researchers have used different measures of excesses like excessive volumes of temporary objects and excessive data copies to recognize the presence of bloat. For example, Xu et al [31] use an instrumented JVM to profile and summarize chains of runtime data copies, while Dufour et al [9] apply blended static and dynamic analysis techniques to runtime traces for characterizing the usage of temporaries.

Java energy characterization Most prior work on energy (and power) characterization of the Java runtime and applications [4, 10, 15, 27] have been simulator based studies or primarily conducted on embedded platforms. Contreras and Martonosi [7] used real system power measurements (like we do) to obtain a component-wise comparisons of power and energy of the Jikes RVM and the Kaffe virtual machine when running client workloads on a Pentium M vs an embedded Intel XScale platform. Their results show that a significant proportion of energy is consumed by the JVM, particularly components like the classloader, JIT and garbage collector. Our work differs in its focus on the impact of a specific category of bloat on power-performance for a given workload. Further we study this impact on large server systems running a production JVM, where the extent of applicability of many of the previous results is not clear.

Object churn analysis, impact and solutions Compiler and runtime optimizations like escape analysis [2, 6, 11, 29] and improvements in memory management and garbage collection techniques [1] have been developed to reduce the

overheads of allocating and reclaiming temporary objects. However, Shankar et al [24] found that even a sophisticated escape analysis implementation in high performance production JVM typically eliminates less than 10% of allocations in component based applications, which matches our observations. They experimented with the use of aggressive guided inlining of regions with high object churn to enable the JIT to detect more opportunities.

Performance understanding techniques have been proposed [3, 9] for guiding programmers in eliminating excess temporaries that cannot be automatically detected by runtime optimizers. For example Buytaert et al [3] identify locations where code refactoring can be applied to reduce object creations. Escape detection has been proposed as an alternative to escape analysis when objects have a high likelihood of being captured, but a conservative analysis cannot detect these with guaranteed correctness. Other alternatives include advancements in memory management techniques for ensuring faster reclamation or reuse of temporary objects, e.g taking better advantage of allocation phases in the application [8, 30], or combining the benefits of explicit object reuse [5, 14] with garbage collection or scoped batch reclamation.

While these efforts are oriented towards fixing the problem, either through automated solutions or through analysis techniques which help pinpoint the sources of the problem, our work complements these efforts by studying the systems level power-performance impact of reducing and increasing object churn. We believe that we are the first to perform such a study, particularly with real power measurements on large scale server systems with dynamic power management capabilities. This is particularly interesting because an improvement in performance need not necessarily translate to an equivalent improvement in power efficiency on such systems, and power savings may occur without a corresponding increase in performance.

Zhao et al [32] analysed the implications of object allocation on scalability and performance. We were able to reproduce similar results during our experimentation, particularly on the Intel Harperton system which was appears to be closest in configuration to their experimental setup. However, power-efficiency was not a consideration in their work.

8. Conclusions and future work

The high degree of software modularization favored for rapid development introduces software bloat of various forms. On a different front, power consumption of servers has emerged as the dominant hurdle to computing performance growth both from approaching technological limits to squeezing increasing performance from a given power budget as well as the rapid growth in energy costs of servers and datacenters. In this paper, we explored the intersection of both these areas.

We presented, what is to the best of our knowledge, the first experimental study of the power-performance implications of software bloat. We focused on one specific category of runtime bloat in great detail, the creation of excessive temporary objects and its impact on performance, power and the energy-efficiency metrics for SPECpower_ssj2008, the first commercial power-performance benchmark requiring energy efficiency reports across a full range of system loads. We performed a detailed cross platform evaluation of these implications on 3 different modern servers under different configurations covering different processors, multiple levels of multithreading, cache sizes, power management, JVM tuning and heap sizes. Our results show significant benefits for software bloat reduction, ranging from 7% to 59%. We find that workload tuning and system characteristics such as SMT and cache size both are important factors in the impact of bloat reduction, e.g. non-tuned (out of the box) JVM configurations show more than 20% improvement with lower bloat. Power management features were found to have a synergistic relation with the benefits from bloat reduction, with DVFS achieving 42% power savings between the highest bloat alternative and the original implementation at equivalent performance, compared to 3% savings without DVFS.

We consider this study to be an important first step in exploring the connection between software bloat and energy efficiency. Interesting future directions include the exploitation of higher level semantic insights (like we used in our hand optimization efforts) for automated bloat reduction and analysing the impact of multiple categories of bloat with realistic workloads. Our preliminary experiments with the DaCapo tradesoap and tradebean benchmarks indicate the potential for far more significant improvements in power efficiency from widespread removal of software bloat.

Acknowledgments

We thank Gary Sevitsky, Nick Mitchell, Matt Arnold, Edith Schonberg for their suggestions and insightful discussions about Java runtime bloat and its implications over the last year and a half. Kazuaki Ishigaki shared an analysis of the SpecPower data model and first brought our attention to the problematic allocation site in the SpecPower code. We are particularly grateful to him and to Dibyendu Das who contributed to early discussions that led to this paper.

We also thank Ankita Garg, Vaidyanathan Srinivasan, Balbir Singh and Dipankar Sarma from IBM Linux Technology Center Energy Management team, Prasanna Kalle, Amar Devegowda, Derek Inglis, Nikola Grcevski and Prashanth Nageshappa from the IBM Java Technology Center and Vijay Mann and Venkateshwara Madduri from the IBM India Research Lab for their help during the course of this work.

We thank Hong Hua for her help in tuning SPECpower executions. We thank John Carter, Heather Hanson, Freeman Rawson, Todd Rosedahl, and Malcolm Ware for their col-

laboration with the second author in developing the energy savings techniques used in the explorations on the POWER7 platform. We also thank Charles Lefurgy for leading the infrastrcuture and tools development for our real-time power-performance monitoring needs on the POWER7 platform.

We thank Prof Gopinath from the Indian Institute of Science for his guidance in pursuing this research.

References

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. *SIGPLAN Not.*, 39(10):50–68, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1035292.1028982.

[2] B. Blanchet. Escape analysis for object-oriented languages: application to java. *SIGPLAN Not.*, 34(10):20–34, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/320385.320387.

[3] D. Buytaert, K. Beyls, and K. De Bosschere. Hinting refactorings to reduce object creation in java. In *Proceedings of the fifth ACES Symposium*, pages 73–76, 1 2005.

[4] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Trans. Embed. Comput. Syst.*, 1(1):27–55, 2002. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/581888.581892.

[5] S. Cherem and R. Rugina. Uniqueness inference for compile-time object deallocation. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 117–128, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: http://doi.acm.org/10.1145/1296907.1296923.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34(10):1–19, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/320385.320386.

[7] G. Contreras and M. Martonosi. Techniques for real-system characterization of java virtual machine energy and power behavior. *IEEE Workload Characterization Symposium*, 0: 29–38, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/IISWC.2006.302727.

[8] C. Ding, C. Zhang, X. Shen, and M. Ogihara. Gated memory control for memory monitoring, leak detection and garbage collection. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 62–67, New York, NY, USA, 2005. ACM. ISBN 1-59593-147-3. doi: http://doi.acm.org/10.1145/1111583.1111593.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *SIGSOFT '08/FSE-16*, pages 59–70, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: http://doi.acm.org/10.1145/1453101.1453111.

[10] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. *SIGMETRICS Perform. Eval. Rev.*, 28(1):252–263, 2000. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/345063.339421.

[11] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93, London, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X.

[12] D. Gray, Larry, A. Kumar, and H. H. Li. Workload characterization of the specpower_ssj2008 benchmark. In *SPEC International Performance Evaluation Workshop, SIPEW*, volume 5119/2008, pages 262–282. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-69813-5.

[13] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy efficiency: The new holy grail of data management systems research. In *CIDR*, January 2009.

[14] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 386–396, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542520.

[15] S. Lafond and J. Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007. ISSN 1383-7621. doi: http://dx.doi.org/10.1016/j.sysarc.2006.10.003.

[16] W. Lang and J. M. Patel. Towards eco-friendly database management systems. In *CIDR*, January 2009.

[17] H.-Y. McCreary, M. A. Broyles, M. S. Floyd, A. J. Geissler, S. P. Hartman, F. L. Rawson, T. J. Rosedahl, J. C. Rubio, and M. S. Ware. EnergyScale for IBM POWER6 microprocessor-based systems. *IBM Journal of Research and Development*, 51(6):775–786, November 2007.

[18] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA '07*, pages 245–260, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297046.

[19] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behaviour in framework based applications. In *ECOOP*. SpringerLink, LNCS, 2006.

[20] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *ECOOP*. SpringerLink, LNCS, 2009.

[21] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to java runtime bloat. *IEEE Software*, 27(1):56–63, 2010. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2010.7.

[22] K. Rajamani, H. Hanson, M. Ware, J. Carter, and F. Rawson. Slack Detection and Exploitation for Performance-Aware Power Management. Technical Report RC24734, IBM Research, January 2009.

[23] E. Saxe. Power efficient software. *Communications of the ACM*, 53(2):44–48, Feb 2010.

[24] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 127–142, New York, NY, USA, 2008. ACM.

ISBN 978-1-60558-215-3. doi: http://doi.acm.org/10.1145/1449764.1449775.

[25] Standard Performance Evaluation Corporation. http://www.specbench.org.

[26] The DaCapo Benchmark Suite. URL `http://www.dacapobench.org`.

[27] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 23–23, Berkeley, CA, USA, 2001. USENIX Association.

[28] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter. Architecting for power management: The IBM POWER7 approach. In *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA-16)*, January 2010.

[29] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. *SIGPLAN Not.*, 34(10):187–206, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/320385.320400.

[30] F. Xian, W. Srisa-an, and H. Jiang. Microphase: an approach to proactively invoking garbage collection for improved performance. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 77–96, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297034.

[31] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09*, pages 419–430, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542523.

[32] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 361–376, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: http://doi.acm.org/10.1145/1640089.1640116.

[33] Y. T. Zichen Xu and X. Wang. Exploring power-performance tradeoffs in database systems. In *ICDE*, Feb 2010.