Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 109C (2017) 1170–1175

Procedia Computer Science

www.elsevier.com/locate/procedia

International Workshop on Adaptive Technology (WAT 2017)

On the adaptive instantiation of type-specific collections

Bruno Sofiato[a,*], Ricardo Luis de Azevedo Rocha[a]

[a]*Department of Computer Engineering, Escola Politécnica, Universidade de São Paulo*
Av. Luciano Gualberto, travessa 3, 380, 05508-900, Sao Paulo, SP, Brazil

Abstract

After an age of almost unlimited computing resources, the advent of resource-constrained computing devices such as cell phones, brought questions of performance back into the spotlight. Programming languages usually provide a set of general-purpose collections. Albeit highly optimized, they fall short of type-specific implementations in both processing time and memory usage. We describe a method for automatically deciding which implementation to use in runtime. Two complementary strategies for determining whether a type-specific collection should be employed are described. To measure their gains, we conducted an experiment composed of eleven benchmarks, ranging from databased to compilers. Results show improvements of 13% in processing and of 3.5% in memory usage. As an unexpected side-effect, our technique reduced instantiations of collection in runtime.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Java, Adaptive Programming, Collections, Aspect-oriented Programming, Object-oriented Programming, Self-modifying Software

1. Introduction

Since the beginning of computing, the number of available computing resources have steadily increased. Moore's Law [1] predicted that the complexity of an integrated circuitry doubles every 18 months. The sheer amount of computing power led to a paradigm shift: writing maintainable code and improving developer productivity became more desirable than writing resource-conscientious code.

The rapid increase of smartphone usage changed this scenario. Besides having less computing power, these devices run on battery. In the competitive world of mobile applications, energy-hungry application are in disadvantage: it might be frowned upon by users, who may ultimately look for an energy-friendlier alternative. Writing an optimized code became a necessity again.

Data structures, such as collection, are an integral part of almost every modern program. Languages usually ship with their own set of collections, freeing the developer from the burden of writing their own. However, their usage poses another challenge: libraries usually ship with a number of collections, whose performance profile varies subtly. Manotas et al. [2] recognizes this issue and proposes SEEDS: a tool that gathers data at runtime and advises the developer on which collection is more energy-friendly.

* Corresponding author
E-mail addresses: bruno.sofiato@gmail.com (Bruno Sofiato)., rlarocha@usp.br (Ricardo Luis de Azevedo Rocha).

Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2017.05.392

According to Neto[3], an adaptive device changes its behavior based on its input. An adaptive collection could change its structure and algorithms based on its elements. For instance, an integer-specific collection implementation might be used when they are all integers. If a string were included, the collection itself would fall back to a more general implementation.

This paper aims to measure the impacts of adopting adaptive techniques to transparently instantiate type-specific collections whenever possible. We describe two techniques – static and dynamic resolution – to determine whether a type-specific collection should be employed. The latter is built upon the premise that the elements inside a collection share the same type.

The Java language was chosen because of its popularity, being the language of choice for developing Android devices applications. Moreover, the Java rich ecosystem facilitates building the prototype: libraries containing type-specific implementation of collections are readily available. Furthermore, the DaCapo[4] suite is based on Java. DaCapo is a battle-proven benchmark suite composed of a plethora of benchmarks, ranging from databases to compilers. This broad range of benchmarks enables results to reflect the expected real-world performance. The DaCapo suite (version 2009) serves as a basis for an experiment, in which we measure the impacts of the techniques presented.

The remainder of this paper is structured as follows: a discussion on the benefits of type-specific collections and on the techniques for deciding upon their usage; an outline of the internals of the prototype built; a description of the experiment and a result analysis. We also present a summary of related works and possible developments of this research. Lastly, we provide a brief conclusion on our findings.

2. Type-specific Collections

Most mainstream languages ship with a general-purpose collection library. These libraries are usually well-known among developers and provide a wide range of highly optimized collections, including linked lists, hash tables and sets. Both C++ and Java ship with their own collection framework – the C++ Standard Template Library[5] (STL) and Java Collection Framework[6] (JCF).

Given their prevalence, it is desirable for those collections to be as efficient as possible. One possible optimization is to have distinct collection implementations based on their element's type. Using type-based collections may yield both memory usage and execution time improvements. For example, a clever implementation of a list may use bitmasks to minimize Boolean storage.

STL collections relies on a C++ construct named templates. Roughly, templates are parameterized blueprints from which the compiler generates new classes. Templates may also be specialized – for a given parameter it is possible to provide a completely distinct implementation, which is transparently resolved by the compiler. Template specialization enables the instantiation of type-specific collections. In fact, some of the STL collections have type-specific implementations.

JFC collections are built upon generics instead of templates. Albeit similar from the developer's viewpoint, generics and templates operate differently. The former does not yield new classes for every parameterization set. Instead, parameters are used solely for type checking. After compilation, any typing information is removed from the resulting byte-code in a process called type-erasure[6]. Moreover, generics do not provide any form of type-based specialization. These characteristics prevent the transparent instantiation of type-specific collections in Java.

3. Adaptive instantiation of collection

There are several Java third-party libraries[7,8] providing type-specific collections. Unlike STL, which transparently handles the instantiation of type-specific collections, these libraries usually define a new set of classes for each Java primitive type, being up to the developer to explicitly instantiate them.

Adaptive techniques could be employed to decide, for every collection instantiation, whether a type-specific implementation should be employed. This approach frees the developer from the burden of choosing the right implementation.

Two adaptive techniques to decide upon the type-specific collection usage are described. The first can only be used by statically-typed languages, whereas the second can be used by statically and dynamic-typed languages alike. They are not mutually exclusive. In fact, they are used together in one of the experiment scenarios.

3.1. Static resolution

Static resolution is based on the type-safety of Java generics. Once they are employed, the compiler prevents any attempts of including elements from other types by raising compilations errors. In such cases, it is therefore safe to replace a regular implementation by a type-specific one, given that their methods share the same semantics.

Notably, static resolution is not strictly an adaptive technique: there are no behavioral changes after deciding upon a particular type-specific implementation However, the described prototype (Section 4) uses adaptive techniques to implement the static resolution: albeit attainable during compilation, the decision of employing a type-specific collection is postpone until runtime. Effectively, the program changes its behavior during its execution.

3.2. Dynamic resolution

In dynamic resolution, the decision on which implementation to use is postponed until its first element is inserted. Depending on the type of the first element, a type-specific implementation can be employed. At each insertion, the type of the soon-to-be-inserted element is checked. If a type mismatch is detected, the runtime falls back to a non-type-specific collection. otherwise, the element is simply inserted.

Falling back to a non-type-specific implementation is an expensive operation. Besides creating the new collection itself, inserting every element from the old collection into the new one is required. Furthermore, both collections must remain in the memory during the copy, potentially doubling the amount of used memory.

Nonetheless, dynamic resolution can still yield improvements. Intuitively, one may infer that elements inside a collection tend to be of a single type, otherwise, type-casting errors would be more common. If this hypothesis holds, then fallback operations are rarely performed and therefore play a minor role on the overall performance.

4. Prototype

In order to measure the impacts of the aforementioned resolution techniques, an executable prototype is necessary. Notably, the described prototype is not meant for production usage.

4.1. Type-specific collections

Creating a clean-room implementation of type-specific collections was not desirable. Beside being time-consuming and error-prone, this implementation may lack optimization. This could lead to skewed results that could minimize the gains from using type-specific collection. Hence, a third-party library was chosen to provide the prototype type-specific collections.

Fastutils[7] was the library chosen. Created by the UbiCrawler[9] team, Fastutils provides several highly optimized data-structures. Among them are type-specific implementation of the most common JCF data structures. Fastutils was chosen based on its rich set of type-specific collections – almost every Java primitive value has its own type-specific implementation of the most widely used JCF collections. Furthermore, they are drop-in replacements (i.e. both realize the same interfaces and their operations share the same semantics), avoiding the need of writing adapters.

4.2. Implementing conditional instantiation of collections

Aspect-Oriented Programming[10] (AOP) techniques are employed to intercept constructor calls. AOP enables arbitrary code (advices) to be injected (weaved) at specific points of execution (pointcuts). Despite not natively supported by the Java platform, several libraries enable the usage of AOP in Java. One is AspectJ, which provides the prototype AOP facilities.

One aspect for each of the selected[1] JFC collections is defined. They intercept the underlying collection constructor calls and return type-specific instances whenever possible. AspectJ Load-Time Weaver (LTW) weaves the aspects. LTW does not require any soon-to-be-woven class to be recompiled, which facilitates its usage with DaCapo.

[1] *ArrayList*, *HashMap*, *LinkedHashMap*, *HashSet* and *LinkedHashSet*.

The way Java platform handles constructor calls prevents simply returning a Fastutils' collection from inside a JFC collection constructor. The compiler introduces byte-code which casts the value returned by a constructor to its declaring class. None of Fastutil's type-specific collections extends from their JFC counterparts. Hence, *ClassCastExceptions* are raised.

To work around this idiosyncrasy, proxies are returned instead of the actual instances. These proxies formally extend from the JFC classes they are replacing, hence eliminating *ClassCastExceptions*. Each of their methods solely calls its counterpart in the proxied collection. The only exception is the *clone* method, which clones the proxied collection and returns a new proxy.

4.3. Collection's type reification

As previously noted, generics type information is unavailable at runtime. However, this information is required for choosing which type-specific implementation to instantiate. To reify the lost type information, the prototype employs a novel technique: both source and byte-code are packaged together and the prototype inspects the former to obtain type information at runtime.

Albeit capable of reifying the parameterized type of collections, this technique is not recommended for production usage. The process of reading the source-code, parsing it and searching for typing information is time-consuming. Moreover, embedding the source-code within the executable package may be forbidden for security reasons.

5. Experiment

We conducted an experiment based on the DaCapo Suite measuring the gains of adopting adaptive techniques to employ type-specific collections whenever possible. The DaCapo Suite was modified[2] to allow using the prototype. These changes were restricted to the packaging of both the prototype and the source-code of each benchmark along the suite (the latter required by the collection-type reification technique described in Section 4.3). Only three benchmarks – *tradebeans*, *tradesoap* and *jython* – were left out. The first two are based on Apache Geronimo[11], whose class-loading scheme prevents LTW whilst the latter is heavily based on bytecode manipulation, which is impervious to our technique. Due to a bug as to how tomcat compares auto-boxed values, adaptive collection instantiation had to be disabled in one of its classes (*org.apache.catalina.startup.Bootstrap*).

Three distinct execution scenarios were defined: *baseline*, where no type-specific collection is employed; *static*, in which only the static resolution is enabled; and *static + dynamic*, where both static and dynamic resolution are used simultaneously. All operations are executed, regardless of the techniques being employed by any given scenario (e.g. the source-code parsing is executed even when running the baseline). This ensured that only the changes due to the employed resolution technique impact the measurement.

5.1. Metrics

Two metrics are defined to evaluate the impacts on performance by using type-specific collections. They deal with two distinct aspects of performance – CPU and memory usage. Although none of the metrics directly deals with energy efficiency, one could extrapolate that an improvement of processing time would lead to an energy usage drop, since there is a correlation[12] between CPU usage and power consumption.

The normalized execution time (NET) is the ratio between the time spent during a benchmark iteration and the baseline's. The median of the first five samples within a 0.75% variance is assumed.

Memory pressure (MP) gauges how much of the memory subsystem is stressed during each sampling. The memory pressure of a given scenario is the ratio between the rate in which the garbage collector reclaims memory (in bytes per second) and baseline's.

Both the number of fallback operations and the general/type-specific instances ratio are collected. They ensure a better understanding of the collection usage profile of each benchmark and allow verifying the hypothesis that collections usually hold elements of a single type (Section 3.2).

[2] A package containing both the prototype and the modified DaCapo is available at `http://tinyurl.com/zextama`.

5.2. Results

Table 1. Results.

	Baseline	Static			Static + Dynamic				
Benchmark	Instances	NET	MP	T. Specific	NET	MP	T. Specific	C. Instances	Fallback
avrora	236	0.9712	0.9998	0.00%	0.9727	0.9997	1.40%	91.10%	0.00%
batik	21026	0.9887	0.9999	0.00%	0.8980	0.9508	6.95%	39.57%	4.86%
eclipse	18800	0.9895	0.9939	0.00%	0.9902	0.9935	6.29%	74.96%	5.83%
fop	77681	1.0000	1.0002	0.00%	0.6618	0.7241	9.84%	69.21%	0.13%
h2	1848536	1.0225	0.6662	6.39%	1.0178	1.0597	6.39%	100.00%	0.00%
luindex	1360	0.9858	0.7539	0.00%	1.0028	1.0768	0.07%	98.82%	0.00%
lusearch	631757	0.9971	0.9996	0.00%	0.1935	0.9572	0.00%	99.85%	0.00%
pmd	295451	1.0135	1.0110	0.19%	0.9948	1.0183	0.20%	99.98%	0.00%
sunflow	2	0.9917	0.9999	0.00%	0.9988	0.9999	0.00%	100.00%	0.00%
tomcat	381448	1.0000	1.0000	0.00%	0.8835	0.9170	0.60%	45.20%	0.00%
xalan	163585	0.9985	0.9986	0.00%	0.9611	0.9857	0.02%	42.05%	0.00%

Execution environment: i7-4790K (3,6 Ghz) with 32GB de RAM, running Windows 10. JDK 1.8.0_92 (64-bits). 256Mb of heap (*-Xms256m* and *-Xmx256m*). Serial garbage collector(*-XX:+UseSerialGC*). Disabled just-in-time (JIT) compiler (*-Xint*).

Table 1 presents the experiment results. On average, we measured an improvement of about 13% in execution time and of 3.5% in memory pressure. *H2* appeared to have a worst overall performance when static and dynamic resolution were employed together. Further analysis revealed that it was due to all of its created collections containing at least one element during execution, which required both the proxy and the collection itself to be stored in memory.

Contrary to our initial hypothesis, fallbacks do occur. In some cases – *batik* and *eclipse* – about 5% of the collections fell back. Such occurrences did not seem to have a detrimental effect on performance. We suspect that this is due to the collection falling back at the beginning of its life-cycle, when it still has few elements. Likewise, the general/type-specific instance ratio was lower than expected: 3% on average. Nonetheless, the benchmark with the highest type-specific ratio – *fop* – showed the most improvements.

We also observed an interesting side-effect: fewer collections are created in runtime when dynamic resolution is employed. Further analysis reveals that this is due to the actual collection creation being postponed until the first elemens insertion. On average, 22% (up to 60% in *batik* case) of the collections remained empty during execution and were hence not created. This behavior played a major role in performance improvement. Almost every benchmark presenting a reduction of collection instances also exhibited performance improvements, the exception being *luindex*.

6. Related Work

Chameleon[13] focuses on space optimizations and employs a distinctive approach. It analyses the collections usage profile in run-time, and, after the execution finishes, it presents a report describing changes. It is up to the developer to decide whether the changes are accepted or rejected. Experimental results show improvements in both processing time and memory usage. Unlike our proposal, Chameleon does not employ type-specific implementations.

Both Manotas et al.[2] and Pereira et al.[14] focus on the power consumption of JFC collections. The former describes SEEDS, a tool that suggests code changes to optimize energy usage whereas the latter depicts the power consumption profile of JFC's collections down to the method level. Manotas et al. claim that an improvement of 17% on energy consumption was attainable by following the SEEDS suggestions.

Sutter et al.[15] present a technique in which type constraints and profiling information are used for suggesting customizations to JFC collections. The former determines when it is safe to employ a specific implementation whilst the latter gauges its gains. Source code changes are required to adopt these suggestions, whereas ours enhances the byte-code of the program when loading. Sutter et al. measured speedups averaging 24% when using this technique.

Contrary to our proposal, these proposals require changes at the source-code level to leverage their gains. This characteristic alone precludes their usage within classes whose source-code is unavailable (e.g. third-party libraries and frameworks).

7. Future Work

A viable development of this research is incorporating both Chameleon and SEEDS techniques for choosing the optimal collection into our approach. An analysis of the impacts of each technique is necessary. The dependency upon IBM JVM must also be addressed. Another approach is to extend both to employ type-specific collections.

As previously noted, the prototype employs techniques which are unsuitable for production usage. Furthermore, the proxy-based solution is neither time nor space-wise optimal. The Valhalla Project[16] aims to incorporate both reified and type-specialized generics into the Java platform. Akin to C++'s template specialization, the latter would enable the creation of type-specific implementations. As these features are introduced, we plan to build a production-ready prototype and to check how it fares against the unmodified benchmarks.

8. Conclusions

By employing both type-specific collections and adaptive techniques, we achieved improvements averaging 13% in processing time and 3.5% in memory usage without any changes in the source-code. Most gains came from combining static and dynamic resolution, the latter being achievable only by using adaptive techniques. We also concluded that fallback operations do occur in practice but they do not hurt performance.

We also observed a scenario on which employing the dynamic resolution hindered performance. In this particular scenario every collection was properly annotated with its element type. Moreover, every created collection instance held at least one element. In such cases, employing the static resolution alone yields better performance.

Nonetheless, we believe that an adequate platform-level support is paramount for a production-ready implementation. In the Java case, both generic-type reification[16] and changes in how the compiler issues byte-code dealing with constructor calls are necessary.

References

1. Moore, G.E.. Readings in computer architecture. chap. Cramming More Components Onto Integrated Circuits. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1-55860-539-8; 2000, p. 56–59.
2. Manotas, I., Pollock, L., Clause, J.. Seeds: a software engineer's energy-optimization decision support framework. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM; 2014, p. 503–514.
3. Neto, J.J.. *Adaptive Rule-Driven Devices - General Formulation and Case Study*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-36390-3; 2002, p. 234–250. doi:\bibinfo{doi}{10.1007/3-540-36390-4_20}. URL `http://dx.doi.org/10.1007/3-540-36390-4_20`.
4. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press; 2006, p. 169–190.
5. Musser, D.R., Derge, G.J., Saini, A.. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional; 3rd ed.; 2009. ISBN 0321702123, 9780321702128.
6. Naftalin, M., Wadler, P.. *Java generics and collections*. Sebastopol, CA, EUA: O'Reilly Media, Inc.; 2007.
7. Vigna, S.. Fastutil: Fast & compact type-specific collections for java. 2016. URL `http://fastutil.di.unimi.it/`.
8. Eden, R., Parent, J., Randall, J., Friedman, E.. Trove – high perfomance collections for java. 2016. URL `http://trove.starlight-systems.com/`.
9. Boldi, P., Codenotti, B., Santini, M., Vigna, S.. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience* 2004;**34**(8):711–726.
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., et al. Aspect-oriented programming. In: Akşit, M., Matsuoka, S., editors. *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-69127-3; 1997, p. 220–242. doi:\bibinfo{doi}{10.1007/BFb0053381}. URL `http://dx.doi.org/10.1007/BFb0053381`.
11. Foundation, A.S.. Apache geronimo. 2016. URL `http://geronimo.apache.org/`.
12. Blackburn, M., Grid, G.. Five ways to reduce data center server power consumption. *The Green Grid* 2008;**42**:12.
13. Shacham, O., Vechev, M., Yahav, E.. Chameleon: Adaptive selection of collections. *SIGPLAN Not* 2009;**44**(6):408–418. doi:\bibinfo{doi}{10.1145/1543135.1542522}. URL `http://doi.acm.org/10.1145/1543135.1542522`.
14. Pereira, R., Couto, M., Saraiva, J., Cunha, J., Fernandes, J.P.. The influence of the java collection framework on overall energy consumption. In: *Proceedings of the 5th International Workshop on Green and Sustainable Software*. ACM; 2016, p. 15–21.
15. De Sutter, B., Tip, F., Dolby, J.. Customization of java library classes using type constraints and profile information. In: *European Conference on Object-Oriented Programming*. Springer; 2004, p. 584–608.
16. The valhalla project. 2016. URL `http://openjdk.java.net/projects/valhalla/`.