Characterizing Performance and Cache Impacts of Code Multi-Versioning on Multicore Architectures

Peter Zangerl, Peter Thoman, Thomas Fahringer
University of Innsbruck, Institute of Computer Science,
Technikerstrasse 21a, Innsbruck, Austria
{peterz,petert,tf}@dps.uibk.ac.at

***Abstract*—Code multi-versioning is an increasingly widely adopted tool for implementing optimizations which respond to unknown or dynamically changing runtime conditions, without the performance overhead of just-in-time compilation. A common concern in its use is instruction cache performance, due to larger binary sizes increasing cache pressure on the one hand and more unpredictable branching on the other.**

Despite this ongoing interest, there has been no comprehensive study of the impact of multi-versioning so far – particularly in a multi-threaded setting. In this paper, we present a categorization of the parameter space potentially affecting multi-versioned performance, a toolset for exploring this space, and an in-depth characterization of three hardware platforms using this toolset.

I. Introduction

Over the past years, there has been an increasing interest in adapting program optimizations to runtime conditions, particularly for parallel systems. These runtime conditions might include factors unknown at compile time, the presence or absence of external load on a system, or shifting optimization priorities e.g. due to the battery state of an embedded device.

Three common methods exist for enabling this type of runtime adaptivity: 1) Program-level and runtime system flags and parameters, 2) Just-in-time compilation, and 3) Compile-time multi-versioning with dynamic version selection at runtime.

Option 1 provides the flexibility of covering a practically unlimited parameter space by varying combinations of these parameters, however, it is infeasible for many purposes. For example, varying the loop unrolling factor of a hot loop nest is well-known to be an effective optimization [1] which can not be performed without actually generating code implementing it. Even in cases which can be parameterized in principle, such as tiling factors, introducing a dependency on runtime parameters might prevent a compiler from performing important optimizations such as vectorization.

Just-in-time (JIT) compilation, listed as option 2, addresses these issues in the most straightforward manner, by moving at least part of the compilation process to the program execution time. While this approach offers the most complete space of possible optimizations and adaptations, it comes with the cost of compilation times affecting execution time. Due to this fact, high-end compiler optimization and analysis is only viable for very long-running code bases. Recent research approaches, such as performing compilation asynchronously with program execution [2], improve the situation, but can still only mitigate and not eliminate the performance impact. Furthermore, JIT compilation approaches might be precluded on many consumer, embedded or HPC platforms either due to security concerns or issues with distributing the entire toolchain required.

Option 3, compile-time multi-versioning with dynamic version selection at runtime, is an attempt at attaining many of the advantages of JIT compilation with a significantly smaller overhead during program execution, and no additional complexities in program distribution or security vetting. As we will illustrate with a short survey in Section II, this approach is broadly used in current practice and research.

A common point of discussion in the community regarding these multi-versioning approaches with runtime version selection are their practical limits, as well as if and to what extent they impact performance on modern hardware when approaching these limits. Of particular interest in this context is the relation between the number and size of the generated versions and the pressure on the instruction caches of a given hardware platform. Despite these concerns, to the best of our knowledge, there has been no comprehensive study so far which analyses the performance impact of multi-versioning across execution scenarios and hardware architectures in depth. Furthermore, the relationship between hardware parallelism implementations such as multi-core or simultaneous hardware multi-threading (SMT) and multi-versioning remains largely unexplored.

In this paper, we will provide a comprehensive analysis of the topic of code multi-versioning and how it affects performance across a wide variety of scenarios and hardware architectures. Our concrete contributions are as follows:

- A definition and categorization of the parameter space potentially affecting multi-versioning performance.
- A set of utilities to explore and evaluate multi-versioning impacts in this parameter space.
- In-depth characterization of the actual performance impact, both in terms of wall time and highly relevant CPU metrics across three distinct hardware platforms and the complete parameter space identified previously.

The remainder of this paper is structured as follows. Section II will provide an overview of related work and illustrate the importance of multi-versioning in current program optimization research. Section III defines and justifies the parameter space of code multi-versioning, as well as clarifying our method of exploring it. In Section IV we apply this method to a set of target platforms and interpret the obtained results.

II. Related Work

The wide applicability of code multi-versioning as a tool to improve program performance has been demonstrated in a

large body of work over the past two decades, of which we only describe a representative subset.

Numerous publications investigate various uses of this technique to generate multiple versions of program functions with different non-functional properties at compile time and then select the one which best matches the user's preference to include in the final version for a given target architecture [3].

Regarding performing the selection among different variants of a multi-versioned code fragment, there is a variety of approaches. Some employ machine learning [4], while others prune parts of the search space to find good candidates more quickly [5]. Diniz et al. [6] describe a version selection scheme which continuously adapts to changes in the environment.

Zhou et al. [7] present a space-efficient multi-versioning algorithm targeting large code bases and space limited environments, focusing on good performance improvements with only small size increase. They do not investigate the impact of code size increases on performance and cache behavior in detail, and target primarily space-restricted embedded systems rather than multi-core parallel environments.

Thoman et al. [8] deal with automatic multi-versioning for task granularity control in parallel programs. They also sketch the impact of large numbers of small versions in respect to their overhead on a single hardware platform, but lack a comprehensive evaluation of the parameter space across different version selection strategies and platforms.

Multi-versioning or recompilation of code fragments and adapting to the environment as well as program inputs can also be done continuously as shown by [9] and others. The code of an application gets improved by monitoring its execution behavior and feeding back this information to the compiler, which can then generate more specialized versions tailored for the current environment. This process can also be carried out during the run-time of the program where hot parts of the application get optimized more aggressively and replaced directly [10]. As discussed in the introduction, such JIT compilation approaches come with their own set of drawbacks, many of which can be mitigated or eliminated by offline multi-versioning and online version selection.

III. Method

A. Parameter Space

In order to fully characterize the potential impact of multi-versioning on program performance, we have identified a set of parameters which may influence the performance and cache behavior of a particular multi-versioned code fragment. We designed our experimental setup such that each of these parameters can be explored individually, as well as having the option of analyzing how arbitrary combinations of them affect the execution behavior.

In the following we explain the parameter space that may influence the execution behavior of multi-versioned programs:

Version selection strategy How the runtime system selects which version of a multi-versioned code fragment to run is an important aspect and has a considerable influence on the cache impact of an application. Our experimental setup enables us to simulate different strategies employed in practice, also including the two extremes – always executing the same version, as well as selecting a random version every time.

CPU architecture and cache properties The properties of the CPU's caches – especially their size and whether they are dedicated or shared between different cores – affect the runtime behavior of a multi-versioned program. We performed our experiments on three different hardware platforms with different cache properties to investigate their effects on the execution behavior.

Concurrency Depending on the design of the CPU, certain parts are shared between multiple cores and/or hardware threads, and thus there is a potential for resource competition. To observe the effects of these shared resources on the runtime behavior of a multi-versioned code fragment, we performed evaluations for different numbers of threads, where each thread independently runs the same program with the same properties. We also include platforms with no, 2-way and 4-way SMT.

Number of versions The number of versions generated for a certain code fragment is likely to be the most obvious parameter to investigate. A large number of versions increases the size of the generated executable and might lead to an increase in instruction cache misses during program execution.

Code size per version The size of each individual generated function version in the executable also influences the cache behavior of the application. Larger versions increase instruction cache requirements, potentially resulting in more cache misses, however, they also require more time to execute.

Execution time per version Another influential parameter is the time spent in multi-versioned code fragments. Short execution times exacerbate the relative overhead caused by cache misses, while long ones mitigate them. This parameter is indirectly related to the code size of each version, as loops or function calls can increase the execution time of a certain code fragment without taking up additional binary space.

B. Version Generation

In order to evaluate the properties of our parameter space we used a custom toolset based on the Insieme research compiler and runtime system [11]. Our goal was to create a configurable number of versions of a function – all with the same runtime properties such as code size and execution time. The runtime system is then responsible for scheduling the execution of the program being evaluated on the targeted platform with a given number of concurrently running threads.

At runtime, each thread repeatedly performs a given version selection procedure before calling the selected version. The execution of the version selection and the execution of the picked version itself is instrumented in-situ in order to accurately gather the following data even for very short runs: wall time, total CPU time, and instruction cache misses at all levels provided by the hardware.

Listing 1 depicts the basic template used to generate the multi-versioned code fragment for all of our experiments. Our generator can insert an arbitrary number of instructions in the loop of the generated functions as exemplified in lines 3 to 5.

Table I: Evaluation platforms hardware and software setup

System	CPU	Sockets / Cores / Threads	Frequency	Memory	Cache			Software	
					L1d / L1i	L2	L3	OS	Kernel
Intel	Xeon E5-4650	4 / 32 / 64	2.7 GHz	256 GB	32 kB / 32 kB	256 kB	20 MB	CentOS 6.7	2.6.32-573
AMD	Opteron 2435	2 / 12 / 12	2.6 GHz	32 GB	64 kB / 64 kB	512 kB	6 MB	Fedora 19	3.14.27
PowerPC	POWER7 8406-71Y	1 / 8 / 32	3.0 GHz	32 GB	32 kB / 32 kB	256 kB	32 MB	RHEL 6.3	2.6.32-279

```
1  double genFun#{version_id}(double a, double b) {
2    for(int j=0; j<numLoopIterations; ++j) {
3      a *= b + #{version_id};
4      ...
5      a *= b + #{version_id};
6    }
7    return a;
8  }
9
10 int num_versions = #{num_versions};
11 funType funVersions[#{num_versions}] =
12   { genFun1, genFun2, genFun3, genFun4, ... };
```

Listing 1: Code template used for version generation

The execution time consumed by the versions can be adjusted by setting the value of the upper loop bound at runtime and thus – combined with the number of instructions inside the loop – the size and execution time of our generated versions can be adjusted almost arbitrarily. A function pointer for each generated version is inserted into an array, which enables a fast and constant overhead version selection at runtime.

While the amount of work to be done is the same for all versions, the actual calculation performed in the loop body is purposefully distinguished in each version by the addition of the version identifier. This is necessary to ensure that each generated version is sufficiently different from all others to prevent the compiler from optimizing any of them away or reusing code.

IV. Characterization

We obtained measurement data across three hardware platforms, the specifications of which are summarized in Table I. Across all platforms we used GCC 5.1.0 with -O3 optimizations to replicate a realistic production scenario. PAPI 5.4.0 [12] was used to obtain CPU counter measurements. For parallel execution, the thread affinity in all runs was fixed using a fill-socket-first policy, in order to improve the reliability of measurements and minimize variance. All reported numbers and figures are based on medians over five runs.

Across our testing hardware platforms and parameter configurations, we generated 302 400 results comprising five measurements (CPU time, wall time, L1-L3 instruction cache misses) each. All of these have informed our evaluation and analysis in this section, and the selection of heatmaps presented within the paper is chosen in order to illustrate the most important effects for each topic being discussed. The code generation, evaluation, and image generation toolset as well as the binary result repository are available online[1].

1) Single-threaded Cache Misses: We begin this evaluation with a basic sanity check of our toolset. Figure 1 illustrates

[1]https://github.com/peterz-dps/multiversioning-eval

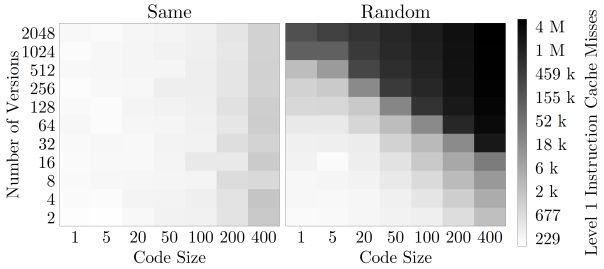

Figure 1: Intel system L1i cache misses, single-threaded.

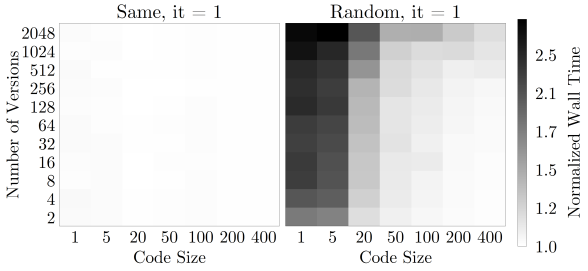

Figure 2: Intel system normalized wall time, single-threaded.

the level 1 instruction cache misses measured on the *Intel* system across a variety of version counts and code sizes, when either always selecting the same version (*Same*) or selecting a random version every time (*Random*). Note that both axes as well as the color coding are logarithmic, and that the range between the minimum and maximum number of cache misses is over 5 orders of magnitude. Two points are worthy of note: (i) If the same version is chosen, the number of cache misses is completely independent of the total number of versions generated. (ii) With a random choice, both a larger selection of versions and larger code sizes lead to a significant increase in instruction cache misses. Plotting L2 instruction cache misses produces a very similar result, with smaller totals overall and the front of significant increases with random selection displaced to the upper right. The *PowerPC* and *AMD* platforms behave similarly, though they top out at a lower number of maximum cache misses.

2) Single-threaded Wall Time: While cache misses are a very useful metric for validating our approach and assumptions, as well as to serve as an explanation for execution time results, those execution times are the most important factor when judging the potential performance impact of a multi-versioning method. Figure 2 depicts the normalized wall time measured across the same space illustrated in Figure 1. The normalization is performed in respect to the execution time of the single-version configuration in each column, in order to eliminate effects unrelated to multi-versioning. A noteworthy

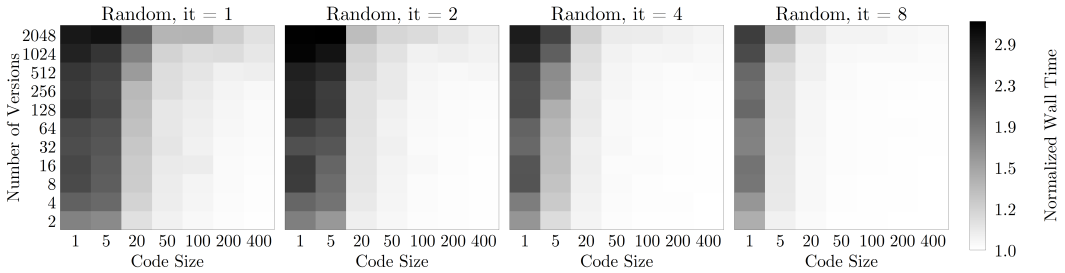

Figure 3: Intel system normalized wall time, single-threaded, across varying inner iteration counts.

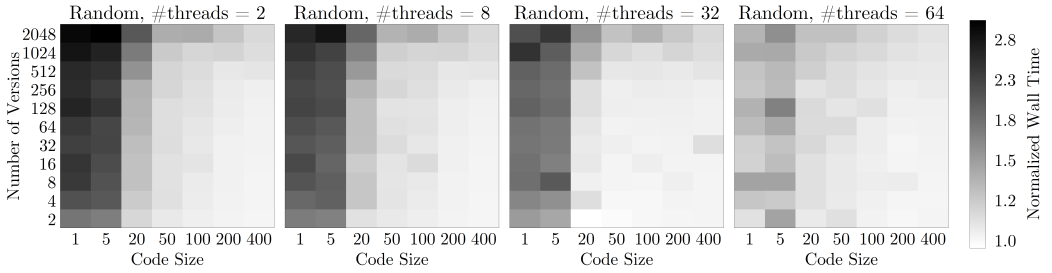

Figure 4: Intel system normalized wall time, single inner iteration, across varying numbers of threads.

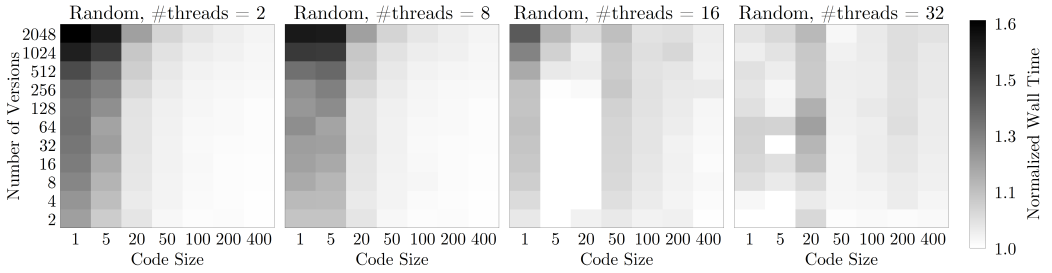

Figure 5: PowerPC system normalized wall time, two inner iterations, across varying numbers of threads.

result which might not be apparent at first glance is that the total range of values only spans a factor of three, despite a difference in cache misses by several orders of magnitude.

For the *Same* selection policy, as the L1 misses would indicate, there are no differences at all across different numbers of code versions. With the *Random* policy, performance degradation up to a factor of three can be observed at the same code size, with a large number of versions. This degradation becomes less pronounced with larger code sizes – despite a larger number of total cache misses – as it is mitigated by larger per-function-call execution times.

3) Impact of Evaluation Time per Version: We will now investigate the impact of varying the execution time per multi-versioned function without varying its binary size. In order to isolate this effect, in Figure 3, four different values for the `numLoopIterations` variable described in Section III-B were chosen, labeled *it*. Given a random selection policy, we observe that with increasing execution time, the relative performance impact of multi-versioning shrinks linearly. This occurs because the instruction cache miss rate remains relatively consistent, especially for larger code sizes and version counts, while the absolute execution time per function increases.

4) Multi-Versioning Parallel Code: As multi-versioning in an optimization context is often leveraged for parallel programs, we have also investigated the impact of multi-threading on the performance of multi-versioned codes. Figure 4 summarizes the impact of increasing the number of threads on normalized wall times on our *Intel* system. All variants which can be mapped to distinct hardware cores show only negligible differences to the single-threaded cases investigated so far, with slightly lower normalized time differences at 32 threads due to a minor increase in the baseline overhead. Once hardware multi-threading is engaged, the relative impact of random version selection drops significantly. This is due to the fact that, upon an instruction cache miss, the CPU core can still keep most of its functional units occupied by switching to the other hardware thread.

On the *PowerPC* architecture, which features up to 4 hardware threads per core, this behavior is consequently even more pronounced. As Figure 5 illustrates, up to 8 threads the behavior remains consistent. With 2 hardware threads used per core (16 threads total), instruction cache misses are mitigated, except in the very short-running function case with code size 1. Using the full 4 hardware threads per core completely eliminates any statistically significant relationship between the number of versions and the normalized execution time.

As the *AMD* architecture we evaluated does not feature any hardware multi-threading, there were no significant changes

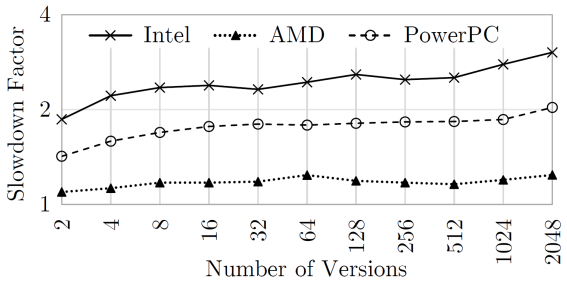

Figure 6: *Random* strategy: worst-case wall time impact.

observed across all viable thread counts, and we omitted this figure for brevity.

5) Hardware Evaluation – Conclusions: From our evaluation of hardware characteristics and multi-versioning behavior so far, we can draw the following conclusions:

- The maximum execution time impact in pathological situations (extremely short multi-versioned code fragments, many versions, random choice each time) reaches a factor of 2 to 3 depending on the hardware platform.
- If the same version is selected throughout the program execution then there is no significant correlation between either cache misses or execution time and the number of code versions. (Disregarding minor fluctuations on one platform related to code alignment)
- While the total number of instruction cache misses increases greatly with any increase in code size, due to the associated increase in execution time of the multi-versioned code fragment the relative impact on execution time actually decreases.
- Increasing the ratio between time spent executing a code fragment and its binary size by e.g. introducing a loop greatly mitigates the relative performance degradation when applying multi-versioning. This is the case even for very small loop iteration counts of a single loop – for longer loops and loop nests the impact becomes completely negligible.
- Multi-versioning a parallel program does not change any of these observations, *unless* a hardware multi-threading architecture is employed. These can leverage the additional threads in flight to reduce the impact of any multi-versioning-related instruction cache misses.

Figure 6 compares the worst-case (across all code sizes, thread counts, and execution times per version) slowdown factors incurred by multi-versioning with random version selection across version counts and our three evaluation platforms. As discussed previously, the maximum performance impact reaches a factor of 3.0 in the worst case (with 2048 versions on the *Intel* platform).

It is important to note that the relative impact is largest on the *Intel* platform and lowest on *AMD* overall for two reasons: (i) The base function invocation overhead and execution time per code line is lowest on *Intel*, which makes any increases relatively more significant. (ii) The *AMD* platform features a first level instruction cache which is twice as large as those of the other two hardware architectures.

V. Conclusion

We have presented a definition and categorization of program code and environmental parameters which can potentially affect the performance and cache effects of multi-versioned programs. To explore this parameter space, we created an evaluation toolset capable of determining the influence of each of these parameters individually, as well as in arbitrary combinations. By leveraging this toolset we carried out an in-depth analysis and characterization of multi-versioning performance on three distinct hardware platforms.

One central conclusion we can draw from this characterization is that once a version to use is decided upon, there is no performance penalty for having a large number of versions available in a binary. Another key observation is that the parallel execution of multi-versioned functions in general does not affect performance negatively compared to the single-threaded case – on the contrary, for CPUs featuring hardware SMT capabilities, executing multi-versioned code in parallel can reduce the overhead by more effectively using the CPU's resources.

Acknowledgement

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603.

References

[1] J. W. Davidson and S. Jinturkar, "Aggressive loop unrolling in a retargetable, optimizing compiler," in *International Conference on Compiler Construction*. Springer, 1996, pp. 59–73.

[2] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 74–85.

[3] K. D. Cooper, M. W. Hall, and K. Kennedy, "Procedure cloning," in *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, Apr 1992, pp. 96–105.

[4] X. Chen and S. Long, "Adaptive multi-versioning for openmp parallelization via machine learning," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, 2009, pp. 907–912.

[5] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam, *A Practical Method for Quickly Evaluating Program Optimizations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 29–46.

[6] P. C. Diniz and M. C. Rinard, "Dynamic feedback: An effective technique for adaptive computing," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 71–84.

[7] M. Zhou, X. Shen, Y. Gao, and G. Yiu, "Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 763–776.

[8] P. Thoman, H. Jordan, and T. Fahringer, "Compiler multiversioning for automatic task granularity control," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 14, pp. 2367–2385, 2014.

[9] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff, "Iterative compilation in program optimization," in *Proc. CPC'10 (Compilers for Parallel Computers)*, 2000, pp. 35–44.

[10] D. R. Engler, "Vcode: A retargetable, extensible, very fast dynamic code generation system," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. New York, NY, USA: ACM, 1996, pp. 160–170.

[11] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[12] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.