Master's Thesis

A PRODUCT-LINE-BASED APPROACH FOR CONSTRUCTING SYNTHETIC SOFTWARE-ANALYSIS BENCHMARKS

GERD ALLIU

July 07, 2023

Advisors:

Florian Sattler	Chair of Software Engineering
Kallistos Weis	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Jan Reineke	Real-Time and Embedded Systems Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Many modern software systems have large numbers of configuration options, which enables them to meet many requirements in a flexible manner. However, the large number of configurations that can be enacted from such options presents a challenge in terms of the analysis of software systems, considering how each or most of a given system's configurations have to be profiled and analyzed for the analysis to be meaningful. To that end, analysis methods have been developed which adequately deal with the complexity that high amounts of configurability introduce, though their development is considerably difficult. In fact, during their development researchers can face significant practical challenges. These challenges can arise due to the lack of access to configurable software systems with adequately documented configurations and certain properties of interest, which would help researchers validate and improve their methods. Moreover, researchers do not have adequate tools that allow them to simulate behaviors of interest in such systems, thus preventing them from being able to address edge cases during their work. Based on these insights, we propose a Product-Line-based approach for producing synthetic benchmarks that match researchers' needs. Crucially, we make use of microarchitectural ideas to ensure that the behaviors of such benchmarks are meaningful and transparent. Overall, we envision that our work will prove valuable to researchers in their quest for better analysis methods, and ignite more interest in using configurability for building benchmarks.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

BPU Branch Prediction Unit

BTB Branch Target Buffer

CPI Cycles per Instruction

CPU Central Processing Unit

CoV Coefficient of Variation

FOSD Feature-Oriented Software Development

GT Ground Truth

ILP Instruction-Level Parallelism

IM Impact-Mapping

IM_{ref} Reference Impact-Mapping

ISG Injective System-Gadget Mapping

IS Instruction Set
LRNG Linear (Congruential) Random Number Generator
MSR Model-Specific Register
NFP Non-Functional Property
PEBS Precise Event-Based Sampling
PHT Pattern History Table
PMU Performance-Monitoring Unit
PRNG Pseudo-Random Number Generator
RAM Random Access Memory
RNG Random Number Generator
SPL Software Product Line
TLB Translation Lookaside Buffer

1 INTRODUCTION

1.1 CONTEXT

Many modern software systems comprise a large number of configuration options, typically in the hundreds [1]. These options can enable entire pieces of functionality in the system that constitute *features* of that system. As such, software systems can have multiple features, as well as many valid combinations of these features. Moreover, a combination of features constitutes a standalone *configuration* of the system and represents a variant of the system with distinct characteristics. It is also commonplace to have many features interact with one another - that is, the presence of one feature imposes some often unintended changes in the workings of another feature during program execution. We commonly refer to systems that have these characteristics as *configurable software systems*, and the practice of designing software systems by making use of features in a systematic way is known as Feature-Oriented Software Development (FOSD).

For all software systems, the analysis of their non-functional properties (e.g. performance) is important in order to improve their quality. Configurable software systems are no exception to this, though the large number of features and the presence of interactions complicate their analysis in several ways [2–4]. The gist of the issue is that the performance of any such system should be examined by taking into account that different configurations will typically contribute to different performance profiles, due to the inherent changes in functionality enabled by their constituent features. These profiles can be quantified and translated into a *performance model*, which is a mathematical description of the contribution of each feature to the overall performance of a system. To construct such models, one would typically need to profile several configurations and extract the overall contribution of each feature. Based on this, we could then obtain clear expectations about the performance of any other configuration, simply by looking at the presumably known contribution of each of the features it comprises.

That being said, the challenge in constructing such performance models is to not resort to brute-force methods of profiling each and every possible configuration, before determining the performance profile of each individual feature. This is due to the fact that the more configurations are excluded, the less information and hence accuracy we would have in the model, although the process is much less time-consuming. Moreover, the manner in which features interact can obfuscate the performance that each individual feature attains in isolation. As such, the performance profile of a configuration cannot be reliably obtained by simply combining the profiles of the features that participate in that configuration. Typically, this is addressed by detecting, profiling, and including interactions in the performance model itself, alongside features.

Facing issues of complexity and tractability, analysis methods that go well beyond naive (brute-force) ideas have been developed. To that end, many analysis methods can be categorized into either *black-box* or *white-box*. Black-box approaches often rely on learning

methods in which the performance profiles of only a subset of the configurations of a program are obtained and then used to derive a performance model that can be used to predict the performance of any other configuration [3]. Moreover, a distinguishing factor of black-box approaches is that the source code of the program is not scrutinized (hence the name). By optimizing the learning process, it is possible to use only a sample of all configurations for the building of a performance model. This way the problem of analysing a huge number of configurations is alleviated. In fact, a big part of the feasibility of many analysis methods rests with the sampling strategy being employed, since it is an effective means of dealing with a huge number of available configurations [3, 5].

Alternatively, white-box approaches can make use of the implementation and structure of a program. Their goal is to automatically scrutinize a program's source code with the aim of extracting clues and important context that helps avoid the sampling of configurations that are redundant with regard to the building of a performance model for the program as a whole. While this adds complexity to the analysis, it can result in better results due to the inclusion of more information. Moreover, it can also better detect and account for feature-interactions [6, 7].

On a similar note, identifying the degree to which configurations are responsible for certain properties, without having complete knowledge of the properties of each configuration, is also possible. For instance, Dubslaff et al. [8] have developed a method for making such explications, using notions of causal inference. Based on that method, they can infer sets of features that are responsible (causes) for certain functional or non-functional properties (effects), as well to determine the degree to which the features in these sets are responsible for said properties.

1.2 MOTIVATION

Based on the analysis techniques described thus far, we observe that it is not always easy or reasonable for researchers to search for and examine existing software systems on the lookout for interesting cases of feature-interactions, large numbers of configurations, or interesting program behaviors. Indeed, the manual labor that is typically involved in scrutinizing existing systems plays a crucial role in specifying a baseline understanding of a system. Moreover, researchers oftentimes need to balance between a deep manual inspection of a few systems, as opposed to a more superficial inspection of a large and varied set of systems.

For instance, Velez et al. [6] relied on several existing systems to determine whether their white-box method was able to correctly detect cases of feature-interactions. The choice of the subject systems played an important role in their work - not only were they real-world examples, they also included a reasonable amount of feature-interactions and certain other characteristics that were deemed to be representative of configurable software systems. Furthermore, the authors had to obtain preliminary knowledge of the actual performance profiles of their subject systems, in order to validate the correctness and efficiency of their method in building performance models.

On a related note, Siegmund et al. [3] constructed a learning-based, black-box method for constructing performance-models, while making intelligent use of sampling notions to make their method tractable. To validate their method, they relied on existing systems

with known configuration options, features, and performance characteristics. However, they correctly noted that the choice of such systems could easily represent a threat to the validity of their approach, mostly due to *overfitting* concerns. Indeed, having a limited number of these systems runs the risk of tailoring the analysis to those systems' characteristics. An analysis tool would rather need to be exposed to different types of systems - that is, systems with varying numbers and types of features and feature-interactions - in order for it to be able to effectively analyze any arbitrary system.

Similarly, Dubslaff et al. [8] also utilized several systems with well-known features and performance models. This knowledge allowed them to determine whether their causal-inference method helps in the identification of causes for observed functional or non-functional properties. Notably, having a preliminary understanding of the exact behaviors and intricacies of features beforehand enabled them to assess whether their inference method identifies effects and causes correctly. Moreover, this also enabled them to see whether it was possible to assign varying degrees of responsibility and blame to already identified causes, w.r.t their effects.

Overall, these observations have led us to pinpoint the following practical challenges for researchers:

1. There is currently no easy way of synthesizing configurable software systems that exhibit specific behaviors (non-functional properties), and have other properties of interest such as feature-interactions.

2. Having preliminary knowledge of a configurable software system's feature-wise behavioral profile is currently not possible without prior examination or analysis of said system.

Therefore, our goal in this thesis is to address the aforementioned challenges. We do so by outlining an approach for creating synthetic programs (benchmarks) that exhibit varied, stable and transparent behaviors, on the basis of FOSD and microarchitectural notions. We outline this approach next.

1.3 SOLUTION

To accomplish our goal of enabling researchers to build synthetic programs with desired properties, we rely on notions of configurability and microarchitectures. Microarchitectural considerations directly help us with regard to the synthesis of stable behaviors in our benchmarks. In addition, configurability notions help us facilitate the synthesis of benchmarks with varied behaviors in a structured manner.

Practically speaking, we first construct components with known features and well-defined behaviors, to which we refer as *gadgets*. These components serve as building blocks for the types of benchmarks we intend to create, as they provide us with important traits that enable their modular use in a larger setting. To develop them, we look toward benchmarking literature, focusing specifically on synthetic microbenchmarks [9–12]. These types of benchmarks apply carefully designed workloads to a system with the aim of evaluating specific aspects of that system. In the same spirit, we design our gadgets so that they apply carefully designed workloads onto a system with the aim of simulating scenarios of performance

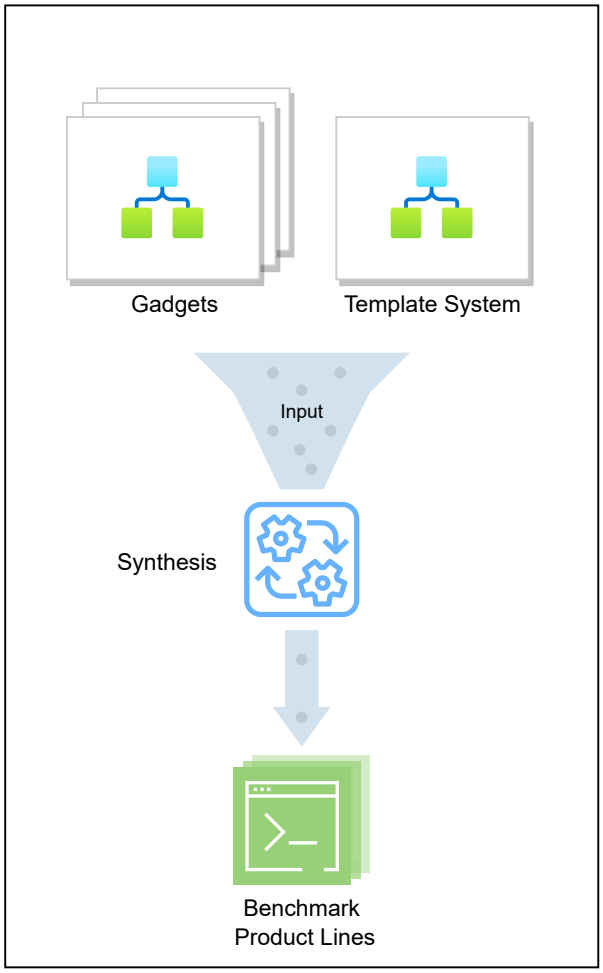

Figure 1.1: A high-level, configurability-based solution concept. Gadgets and a template system are funneled into a synthesis process through which we can produce families of synthetic benchmarks.

regressions, and potentially other behaviors. By design, gadgets offer a small number of features, and therefore a relatively small number of configurations. Importantly, the design of each gadget is such that for each of their configurations we obtain distinct, sizeable, and stable behaviors.

To ensure that our gadgets have stable impacts, we take a resource-focused approach in that we aim to build gadgets that impact specific resources of a system's architecture, such as the Level-3 cache. Firstly, we identify metrics that indicate program behavior from the perspective of an architectural resource. Then, we try to design features that contribute to varied impacts that are noticeable through the chosen metrics. Taking this path enables us to have more control over the behavior of a gadget, which is important for its stability. In addition, focusing on metrics pertaining to low-level resources enables us to quantify and evaluate the behaviors of gadgets effectively. Based on their focused and stable impacts, we make their behavioral profiles transparent to the users.

Nevertheless, gadgets are not the ultimate benchmarks we intend to build. In fact, through the use of gadgets we are able to build synthetic benchmarks with complex behavioral

profiles that are of value to researchers. These synthetic benchmarks are crafted in a way that makes use of gadgets' properties in a careful manner. In practice, this requires some degree of automation and application of ideas such as Software Product Lines (SPLs), which refer to families of products that are crafted on the basis of common artifacts as well as configurability. At a high-level, we implement the gadgets in a well-organized manner, and subsequently combine them under existing configurable systems that act as execution templates. This way, we are able to build entire families of benchmarks that can combine the workloads of multiple gadgets in various ways. The specific manner in which this combination is performed constitutes our primary means of producing synthetic benchmarks and is explained in detail in subsequent chapters. A high-level depiction of our solution is provided in Figure 1.1.

Overall, this approach enables us to produce meaningful benchmarks, in the sense that their behaviors are 1) varied, 2) stable, and consequently 3) transparent, stemming from the fact that the gadgets themselves have stable and transparent behaviors by design, and provide a large variety of impacts due to their inherent configurability. Based on these properties, our approach directly addresses the practical challenges we highlighted in the previous section. In fact, having benchmarks with varied behaviors is a direct result of using configurability, and can help researchers in generalizing their analysis methods by testing them using different benchmarks. Similarly, ensuring stable behaviors in these benchmarks makes them reliable in the sense that they can be used when and where the researchers need to subject them to analysis methods. Moreover, stability allows us to document the behaviors and make the intricate workings of our benchmarks accessible to the researchers, which should help them validate their methods more efficiently.

To realize our approach, we implement a framework that provides all the mechanisms needed to create actual benchmarks. Moreover, we evaluate our implementation and the overall approach in detail. Notably, we ensure that the gadgets have stable and transparent behavioral profiles as intended. In addition, we successfully assess their usefulness in building synthetic benchmarks, while also pinpointing potential limitations and improvements. Overall, we believe that our approach adequately addresses the two practical challenges we outlined, and enables researchers to build test systems that directly aid them in improving and developing analysis methods like the ones we described earlier.

1.4 OVERVIEW

To develop our solution in a coherent manner, we structure the rest of the thesis as follows. In Chapter 2, we provide the necessary background needed to read through the thesis while getting a solid understanding of the main ideas. That includes basic notions of software configurability, computer architecture, and performance engineering. In Chapter 3, we provide a detailed description of our approach, and a formal description of the main concepts. In Chapter 4, we then dive into the implementation details and relevant issues, whereas in Chapter 5 we evaluate key aspects of our work. In Chapter 6, we describe the body of literature that inspired and informed many of our ideas. In conclusion, we summarize our contribution and present future-work ideas in Chapter 7.

2 BACKGROUND

In this chapter, we present important notions of configurable software systems, computer architecture, performance engineering, and benchmarks. All these concepts help in developing the core ideas of gadgets, their stable and transparent behaviors with regard to architectural resources, and ultimately the synthetic benchmarks that can be built based on our approach.

2.1 CONFIGURABLE SOFTWARE SYSTEMS

In this section, we introduce important notions regarding configurable software systems, which we use throughout the thesis. Notably, we introduce the concepts of Feature-Oriented Software Development (FOSD) and Software Product Line (SPL) in some detail, since these concepts are central to our work.

2.1.1 *Configurability & Feature-Oriented Software Development*

Modern software systems meet numerous and complex requirements through their inherent *variability*. At a high level, variability can be understood as the ability of a software system to change its functional or non-functional properties, so that it fits a specific context [13, 14]. To realize variability, a software system or artifact might comprise several variation points, through which a stakeholder can vary its properties. The end result of variability realization is the obtaining of different *variants* of the same software system [14].

One specific type of variation points are the so-called *configuration options*, which typically refer to explicit parameters in the code artifacts of a software system that can be set by a stakeholder. Notably, configuration options are mostly considered with regard to the functional properties of a system and can facilitate the toggling of entire pieces of functionality. Stemming from this idea, the deliberate and structured use of configuration options gives rise to what is known as *configurability*.

Systematically handling configurability in modern software systems can be challenging, especially during development. While it is possible to try to manage configurability through refactoring and better architectural considerations [1], the proactive approach is to bring configurability to the forefront of the engineering process. This is precisely what FOSD achieves. In FOSD, the focus is on *features* - units of functionality that satisfy a requirement, represent a design decision and provide a potential configuration option [15]. Based on this definition, it is possible to develop software systems in which there is a clear mapping between code fragments and logical functionality, regardless of how the code artifacts are organized from an architectural perspective. In addition, logical functionality can be traced from the early stages of domain analysis and design to that of implementation, since the notion of a feature is explicit in the code artifacts. As such, the FOSD paradigm bridges the observable structure and behavior of a system to its implementation.

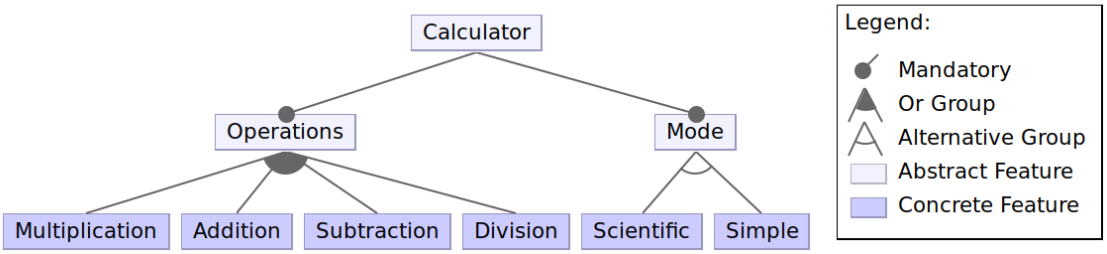

Figure 2.1: Feature Model for a simple Calculator. Stemming from a feature with children features, OR groups indicate that at least one of the child features must be selected, whereas Alternative (XOR) groups indicate that at least and at most one child feature must be selected. A mandatory feature is one that must be selected if and only if its parent is selected. The root feature is always selected.

An important notion in FOSD is that of *feature-models* [15, 16]. A common way of modeling the features of a system, as well as their logical organization and constraints is to depict them in a tree diagram, as shown in Figure 2.1. A hierarchical depiction of this kind places the software system at the root, as an abstract feature that represents the overall functional outline of the system. Features are then organized in this hierarchy based on their conceptual meanings, as well as their inter-dependencies and constraints which are usually depicted using notations for logical constructs such as OR, XOR and AND groups. Moving deeper into the tree we may find other abstract features, though we should typically find concrete features at the leaf-nodes. Such a model is easily amenable to propositional logic and reasoning, since it is easy to derive the entire feature-model of a system in propositional form [16].

The features in such models are usually binary, meaning that they can be either enabled or disabled using certain configuration options. For instance, the simple Calculator program shown in Figure 2.1 could include an Addition feature while not including Multiplication. Non-binary features can also be modelled, however we aim to keep the models simple in this work. Considering a model with only binary features, any group of enabled features constitutes a *configuration* of the software system. In addition, if this group of enabled features respects the constraints and dependencies in the model, it then constitutes a *valid configuration*. Moreover, each valid configuration leads to a variant of the system, which is why these terms are often used interchangeably.

Another important concept in FOSD is that of *feature-interactions*. Kolesnikov et al. [2] define a feature-interaction as the case in which the functional or non-functional properties of a feature are influenced by the presence of another feature in the same configuration. A simple case of two interacting features is one in which the corresponding in-code implementations are structurally interrelated in such a way that the implementation of one feature is affected by the presence of the other. However, features might also interact in non-structural ways, especially with regard to non-functional properties.

In Listing 2.1 we show how the implementation of a Multiplication feature can be affected by that of the Scientific feature, based on the model in Figure 2.1. If the latter is enabled, the function that implements the former will receive a different parameter type which may affect its functional logic.

```
#ifdef MULTPLICATION
    void config_mul(layout* l, representation r) {...}
#endif
.
.
void setup(layout* l){
    .
    r = get_default_representation();
#ifdef MULTIPLICATION
#ifdef SCIENTIFIC
    //Interaction
    r = get_binary_representation();
#endif
    config_mul(l, r);
#endif
    .
}
```

Listing 2.1: Structural Feature Interaction due to control flow. Each feature is enabled on the basis of a preprocessor directive. For instance, if the MULTIPLICATION macro is defined, the code that implements the Multiplication feature will be included during compilation, and therefore in the final executable.

2.1.2 *Software Product Lines*

FOSD provides us with a structured way of handling configurability, from the early stages of domain analysis and design, to the implementation and derivation of different variants. However, in order to further structure the discussion around configurable software systems, we can rely on the notion of a Software Product Line (SPL) [13, 15, 17, 18].

The simplest way of conceptualizing an SPL is to think of all the possible variants that we can derive for a configurable software system, considering its feature-model and configurations. The set of these variants would constitute a *product family*, i.e. a group of different software products that are related to each other in terms of the artifacts they rely on and other commonalities. These artifacts, especially when using FOSD principles, are systematically reused across the entire product family [15, 18]. Besides this notion of systematic reuse, an important idea regarding SPLs is that of decision-making [13, 18]. Naturally, handling an entire set of variants for the same system necessitates decisions such as selecting the desired features to include in a variant, enacting the selection into the code artifacts, and generating (deriving) a final product. The presence of a feature-model greatly helps with the feature selection, though additional tools and methods are usually employed to bind selected features and to perform an automated product derivation.

Overall, the development lifecycle of an SPL is understood in two parts [18], as shown in Figure 2.2. On the top half we depict the Domain Engineering process, in which we conceptualize features and design a feature-model based on overall domain needs (Domain Requirements Engineering & Design), as well as craft code artifacts and introduce configuration options in these artifacts (Domain Realisation & Testing). On the bottom

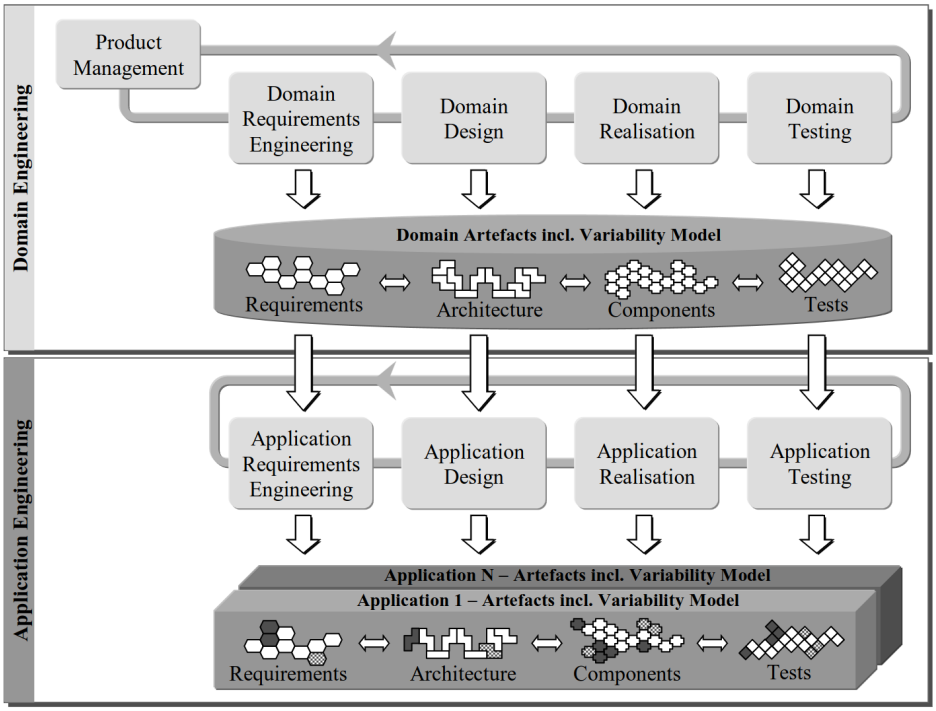

Figure 2.2: SPL processes, adapted from [17].

half we depict the Application Engineering process in which we make feature selections and architectural decisions regarding the composition of features, guided by specific user requirements (Application Requirements Engineering & Design), and derive actual products (Application Realisation & Testing).

Crucially, the development lifecycle of an SPL is contingent on having a clear approach for the transition from Domain Engineering to Application Engineering. The most important part of this transition is that of being able to translate high-level decisions regarding the functionality that a product of our SPL should have, into low-level decisions that allow us to obtain the actual product. Typically, this transition is facilitated through *binding* decisions. As an example, once we have a feature-selection in mind that would give us a variant of a given configurable software system, we need to translate it into specific decisions with regard to the code artifacts and the configuration options thereof [14], so as to obtain the intended variant (product).

In this work, we rely on development-time binding decisions. Technically speaking, we facilitate bindings through compile-time techniques for handling configurability at the code level. For instance, we rely on *preprocessor directives* like the ones in Listing 2.1 to enact a feature-selection and derive a desired product variant. Looking at the code snippet, we can see that the code corresponding to features that we have not selected will simply be omitted from the compilation, whereas the code that corresponds to selected features is included and contributes to a part of the resulting variant's functionality. This way, all that we would have to do to perform a binding is to use the preprocessor directives correctly.

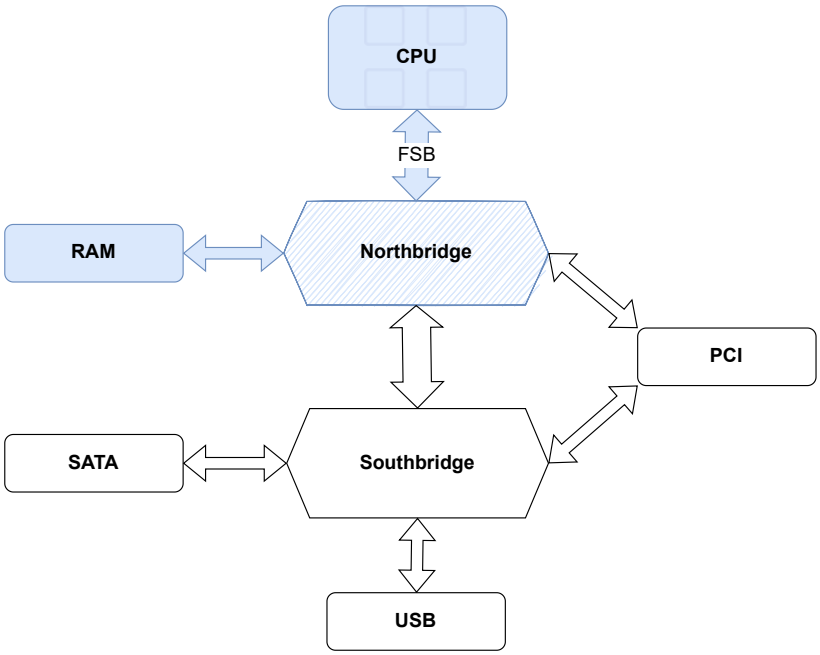

Figure 2.3: Internal organization of a computer system. The parts related to the Northbridge are highlighted in blue.

2.2 COMPUTER ARCHITECTURE

In order for us to discuss microarchitectural ideas related to our approach, we present key notions regarding memory, caching and branch-prediction. These ideas inform our approach and implementation in subsequent chapters, especially w.r.t. the configurable components we call gadgets.

2.2.1 *Basic Notions*

For the goals of this thesis, we need to have a common understanding of computer architecture in place. To facilitate the discussion regarding this concept, we present a simplified depiction of a modern computer system's components and their relations in Figure 2.3. The main building blocks shown in the figure are a single Central Processing Unit (CPU), a Random Access Memory (RAM) unit in the role of the main memory, and the main interconnects that can be conceptually organized into the Northbridge and Southbridge [19, 20]. Notably, the Northbridge encompasses all data buses and components that exist between the main memory and the CPU, whereas the Southbridge encompasses all interconnects between the Northbridge and I/O components. An important takeaway from this description is that there exists a large difference in the workings of different components and interconnections; for instance, the rate of data transfers in the Northbridge is much larger than in the Southbridge (hence the distinction).

Furthermore, the CPU usually consists of multiple cores, as well as other components such as caches that are shared among among cores, and which can store both data and

instructions. Each core is tasked with the fetching and processing of instructions, as well as the storing of the results back to memory. Overall, the internal workings of the CPU, as well as its interfacing with the other components, are guided by the *clock frequency* - each CPU works at the pace of a series of pulses that are generated at a (mostly) fixed frequency. Each pulse initiates a computational cycle across all cores, forcing them to make progress in the processing of instructions.

Considering how instructions and data typically reside in memory and need to be fetched before processing, the internal organization of each core comprises three logical parts: 1) the front-end, 2) the back-end (execution engine), and 3) the memory subsystem. The memory subsystem is the primary interface to the Northbridge, and notably contains the Level-2 cache that is shared by the front-end and back-end of the core and which stores both data and instructions. It also includes the Level-1 data-cache for retrieving and storing data from/into memory. Similarly, the front-end contains the Level-1 instructions-cache to handle the fetching of instructions from memory. Lastly, the back-end incorporates all the components that relate to the actual execution of instructions such as register files and arithmetic logic units, and the buffering of the results prior to sending them to the memory subsystem.

2.2.2 *Memory*

In modern architectures, memory components are organized into conceptual hierarchies relating to their capacities and latencies. As such, we usually see that registers are small yet fast memory units, whereas the caches are progressively larger and slower, and the main memory is the largest and slowest. Practically speaking, accessing data in the main memory can easily take about tens or even hundreds of cycles due to hardware considerations, making for significant processing latencies [19]. The use of multiple levels of caches alleviates this issue, since caches can reduce the number of high-latency accesses to memory, as well as fetch data or instructions from the main memory before they are needed in the CPU [19, 21]. Given this insight, modern computer architectures usually incorporate three[1] levels of caches, typically Levels 1 and 2 which are per core, and Level 3 which is shared among cores. A cache of a higher level is only accessed when the needed data is not found in the cache of the preceding level in what constitutes a *cache miss*, and is typically larger in capacity than its lower level counterparts. For instance, in an Intel Skylake model, the cache sizes for each level can be at most 32, 256 and 2048 Kilobytes, respectively [22]. These are in stark contrast to the sizes of main-memory units which range between 16-256 Gigabytes in most modern systems. Given their limited sizes, caches implement certain policies regarding when and how data need to be inserted, removed and kept.

Since we deal with multi-core CPUs, we usually have several instances of Level-1 and Level-2 caches, whereby each core has a Level-1 and a Level-2 cache. While this is beneficial for performance, cases of inconsistencies between the corresponding per-core caches of the same level could arise when the same memory locations are concerned. For that reason, modern CPUs implement certain algorithms to achieve *cache-coherence*, which refers to the

1 In many cases there is also a fourth level of caching that stores decoded micro-instructions [22], though this is not of direct relevance to this work.

careful coordination that takes place among cores when performing accesses on caches so as to avoid any inconsistencies [23].

Before diving deeper into caches, it is important to have some notions of memory access in place. The storage pattern of a RAM can be described as a two-dimensional array of cells, each of which can store one bit of data. The typical RAM implementation is *dynamic*, which means that the electric state of each cell needs to be periodically refreshed in order for the data to be preserved. This translates to added waiting delays during memory access. To perform an access, the CPU would need to provide an *address* specifying both row and column indices that identify a cell in the array. However, to avoid inherent hardware delays, common implementations operate at a larger granularity than a single cell whereby multiple rows of cells are read at once. In Intel architectures we usually see memory accesses happening in 8 rapid bursts of 8 bytes, usually amounting to a total of 64 bytes which is also known as a *cache-line* due to the fact that caches typically operate in data chunks of that size [19, 20].

Besides the basic idea of memory accesses, it is important to note that modern CPUs rely on virtual addressing [24]. Virtual addresses present a logical space of memory locations that is much larger than the actual physical one offered by a RAM unit. It is the job of the operating system and the CPU to handle physical-to-virtual address translations. The concerned physical-to-virtual mappings are dynamically maintained in specific memory-residing data structures called *page-tables*. Importantly, frequent accesses to the page-tables cause significant latencies, which is why these mappings are also cached in the so-called Translation Lookaside Buffers (TLBs). The working principles for TLBs are quite analogous to those of the main caches [19, 25]. Moreover, TLBs are often integrated with the Level-1 and Level-2 caches. Notably, modern Intel systems usually incorporate two levels TLBs, where both levels are per-core and the first level is typically split into separate units for instruction and data, similar to the standard Level-1 caches [22].

Turning our attention back to caching, the main operating principles that underpin caches are *locality*, *prefetching* and *associativity* [20, 26–28]. Locality refers to the usefulness of data or instructions currently residing in a cache for a future computation. Usually, executing instructions tend to relate to a set of data items that reside close to each-other in terms of their locations. Therefore, when an item from this set is used, the other ones will likely be used next and so they need to reside closer to the CPU (spatial locality). Similarly, data items that were recently used tend to be used again in the near future (temporal locality).

Stemming from ideas of locality, modern CPUs make use of hardware logic to predict what data items or instructions will be needed next based on currently cached data, making sure to fetch those items as early as possible so that they can be immediately used when needed. This technique is known as *prefetching* [26, 27]. Both locality and prefetching help ensure that accesses to the main memory are avoided, which is crucial for performance given the high latencies of accessing the main memory. This implies that when an executing process accesses data (or instructions) in ways that do not adhere well to the locality ideas described previously, its performance can deteriorate. An interesting scenario when this can happen is when prefetching is performed for some data items which are expected to be used shortly after, when in fact that data is not used; indeed, this wastes the effort of prefetching and also uses up some of the cache's capacity in a counterproductive way. Overall, bad use of locality and wasteful prefetches lead to significant numbers of misses in the caches [19].

To provide more context, a miss in the Level-2 caches can cause latencies of more than 10 cycles, and those in Level-3 can cause latencies of about 30-60 cycles[2].

The internal organization of a cache usually tries to balance between having a fixed location for the data residing in each memory address in what is known as *direct-mapped* caching, versus having a free-choice of locations for each address in what is known as *associative* caching. The former strategy results in faster searching but also in a higher miss-rate since the cache can only contain data for a limited number of addresses, whereas the latter can incur latency due to a large numbers of comparisons during the search for a free location, though it attains lower miss-rates. The common solution is to mix both these ideas, resulting in the so called *set-associative* caching [29]. In set-associative caches, the first bits of an address directly map to a set of locations in the cache, with the correct set being identified through some hashing function that takes these bits as input. The middle bits then index into any locations within that set in a fully associative manner, whereas the last bits serve as offsets for identifying actual blocks of data in a specific location. While this approach is widely used nowadays, it can still have some limitations. One potential issue is that many subsequent accesses to a cache can still map to the same set and exhaust its capacity by using up all of its free locations, similar to what happens in direct-mapped caches, even if there are free locations in other sets. Such occurrences are known as *aliasing* effects [19].

2.2.3 *Pipelined Processing*

Besides the concept of computer architecture discussed earlier, an important concept is that of an Instruction Set (IS). An IS specifies the types of instructions that a processor can execute, whereas the actual on-chip implementation of an IS is what is commonly referred to as a *microarchitecture*. In this work we consider only x86 microarchitectures, mostly Intel-based implementations due to their common use in practice.

A major challenge in developing a microarchitecture is for the CPU to process as many instructions as possible per cycle. The processing of any instruction is performed in multiple stages. The main stages are 1) Fetching, 2) Decoding, 3) Executing and 4) Write-Back. As such, one major issue is that given the memory hierarchy discussed earlier, stalls may occur when the execution of an instruction requires memory accesses, thus blocking subsequent instructions from starting to execute. Naturally, some parallelism is required in order to process multiple instructions simultaneously instead of stalling an instruction in one of the stages and blocking the overall flow of instructions. The main form of parallelism that modern microarchitectures use is Instruction-Level Parallelism (ILP) [30].

The core idea behind ILP is that the processing of instructions in the aforementioned stages can be overlapped; for instance, while an instruction is in the Decode stage, another instruction can be in the Fetch stage. This way, the processor can make progress on as many instructions as there are stages, implying that the overall processing can obtain an ideal throughput of one instruction per cycle. This way of processing is known as *pipelining*, and a processor that can obtain a theoretical maximum throughput of one instruction per cycle is called a *scalar* processor [30]. Additionally, pipelines in modern systems are further improved by having concurrently executing instructions for some or all of the four stages

2 Figures are based on reports on Intel's Haswell family of CPUs: https://www.7-cpu.com/cpu/Haswell.html

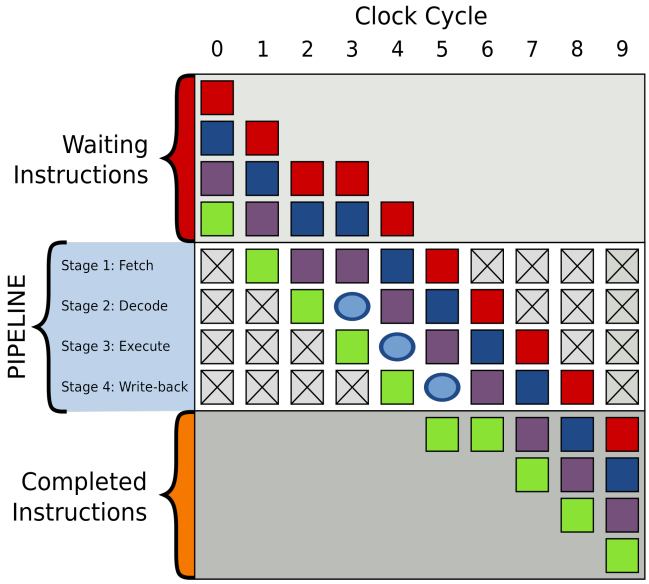

Figure 2.4: Pipeline Bubbles, shown in blue ellipses. Adapted from Wikipedia[3].

mentioned earlier. This implies that it is possible to simultaneously process more than one instruction in each stage, therefore obtaining throughputs beyond one instruction per cycle - such systems are called *superscalar* [30, 31].

While ILP greatly improves system performance, there are some important considerations to note. For instance, branching instructions such as a simple "if" condition in high-level code are challenging because the choice of which instruction to process next depends on whether the branch is taken (condition is true) or not (condition is false). If the processor does not figure out what the next instruction (otherwise known as the target instruction) is before the branch instruction is executed, then it has to initiate a costly memory fetch for that instruction. During this fetch, several pipeline stages can remain empty and thus waste CPU effort, simply because the CPU does not know what new instructions to issue into the pipeline in the mean time. The presence of such stalls in the pipeline are commonly known as bubbles (cf. Figure 2.4) and are costly to performance [30]. Alternatively, if the processor is able to predict the target instruction, then it can continue to do work in which it processes the target and any succeeding instruction, thus wasting no computational effort. Sometime later, the processor will verify whether the prediction was correct, continually trying to improve its future predictions.

In order to reduce the number of bubbles, modern systems incorporate special circuitry to perform branch-predictions in a component known as the Branch Prediction Unit (BPU). These predictions are normally done in a *dynamic* way, i.e. during process execution. Typically, the processor will try to guess whether a branch (e.g. an if-condition) will be taken (condition is true) or not (condition is false). Based on the prediction, it will speculatively process subsequent instructions without stalling. The prediction can be performed before or by the time the branch instruction is decoded in the Decode stage, and its correctness is evaluated when the branch instruction is in the Execute stage. In case of incorrect predictions,

3 https://en.wikipedia.org/w/index.php?title=Pipeline_stall&oldid=1144156605

latencies are incurred due to the fact that the pipeline ends up doing useless work during its speculative execution [32]. Besides the prediction of whether a branch is taken or not, it is also important for the address of the target instruction to be known as early as possible. For simple branch instructions, the target instruction could be known as soon as the taken-or-not prediction is performed, but this might not always be possible. In other words, the target should be obtained as soon as possible, and thus another form of prediction called *target prediction* has to be performed.

To facilitate both types of predictions hand-in-hand, a hardware component known as the Branch Target Buffer (BTB) is used. The BTB is practically a cache that stores previously seen target instructions for a given branch instruction. In addition, the entries in this cache index the so-called Pattern History Tables (PHTs) that maintain histories of predictions for the cached branch instructions. Based on this cache, predictions can happen very early in terms of the pipeline stages. The size of the BTB is thought to be limited to 4096 bytes in Intel architectures [33]. In any case, Intel systems incorporate variants of predictors that use system-wide (global) histories and can thus detect long periodic sequences of branching decisions, resulting in highly accurate predictions overall [33, 34].

2.3 PERFORMANCE MONITORING

To be able to assess the behaviors of any component that we build, we need to establish some key notions regarding performance monitoring. Therefore, in this section we provide an overview of concepts such as metrics, counters and tools that allow us to perform measurements.

2.3.1 *Metrics and Tooling*

In order to build software systems that meet certain non-functional criteria, an understanding of non-functional properties and other performance concepts is required. A program's functional design meets only a part of stakeholders' requirements. The other part is that of providing the functionality in a high quality manner through architectural and performance-engineering decisions [35–38]. Defining quality usually requires the specification of some non-functional attributes, such as the ones shown in Table 2.1. Note that for the most part we are only interested in runtime Non-Functional Properties (NFPs). Certain NFPs that relate to development, testing or maintenance help us in our work, but are not of relevance to this thesis. Furthermore, these properties need to be quantifiable and measurable for them to be useful. Therefore, we need to define some *metrics* for each NFP of interest. In this work, we understand the term metric as a statistic of interest that helps us quantify the behavior of a software system [11].

A common way of performing measurements w.r.t. certain metrics is to rely on the Performance-Monitoring Unit (PMU) of an architecture [20, 28]. All x86 implementations incorporate this unit in their design, providing a reliable way of monitoring and measuring the performance of programs w.r.t. different components such as the Level-1 caches. The PMU comprises several Model-Specific Registers (MSRs), and provides an interface to software

NFP	Type
Performance	Runtime
Reliability	Runtime
Maintainability	Development-Time

Table 2.1: Example NFPs and their types, based on [39].

agents (e.g. perf[4]) for using these registers. In order to obtain a measurement, a software agent can use an MSR to provide identifiers for multiple types of events, and then read corresponding event-occurrence counts in general-purpose registers. Usually, the types of events that can be counted are called *counters*, with multiple of them being available for components in the CPU cores as well as for ones in the Northbridge or Southbridge. Based on counters, (aggregate) metrics of interest can be calculated which we denote as *resource-specific* metrics. The main utility of such metrics is that they provide a very focused view on a program's behavior w.r.t. specific system resources.

Another way of obtaining measurements is *profiling* [28]. Unlike with event-counting, the idea is to periodically interrupt the execution of a process, take a snapshot of the executing code-paths, sample counters or other process-state variables and resume the execution. This method is more obstructive as it interferes with the performance profile of the process subject to the measurements, though it can provide valuable information regarding the code-paths that are responsible for certain behaviors.

2.3.2 *Important Metrics*

Other, non-resource-specific metrics that can help characterize the performance of systems or components are *latency*, *utilization* and *saturation*. The term *latency* is usually convoluted with that of *response-time*; the former refers to the time spent waiting for something to be processed, and the latter refers to the processing time. Regardless, in microarchitectural considerations we find that latency incorporates both the waiting time and response time [30]. Utilization can also be understood in different ways. On one hand, utilization can refer to the time during which a component is busy doing work, in which case it is known as time-based utilization. On the other hand, we can conceive of a case in which a component is busy only 50% of the time, though during that time it is fully occupied with work. In such a case we can say that the component experiences full capacity-based utilization. Lastly, saturation corresponds to a case of complete capacity-based utilization, in which case the component is likely to experience increased processing times and therefore lead to queued work [28].

4 https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=1126307013

2.4 SYNTHETIC BENCHMARKS

The ultimate goal in our thesis is to enable a way of producing benchmarks with certain properties. Therefore, we need to establish some common ground w.r.t. the concept of benchmarking. Kounev, Lange, and Kistowski [11] define a benchmark as a "tool coupled with a methodology for the evaluation and comparison of systems or components with respect to specific characteristics, such as performance, reliability, or security". From this definition, we can discern that a benchmark program aims to operate in a way that exposes some behaviors or traits of some target system (or component). These behaviors can be observed and quantified using the notions discussed previously, whereas their elicitation depends on the workload that the benchmark will apply on the target system.

Based on the scope of evaluation, benchmarks can be classified into *microbenchmarks* or *macrobenchmarks* [11, 28]. Microbenchmarks are of relevance to this work since we try to focus our discussion on one specific architectural component at a time. Another way of classifying benchmarks is to look at how realistic the presented workload is. In order to perform a precise evaluation of the performance of a system or component, it is important to employ a realistic workload. However, realistic workloads are usually presented by real, heavy-duty software, the availability of which is not always possible. To address this issue, a synthetic mix of simple sets of operations can be used, in which case the benchmark is known as a *synthetic benchmark*. The choices that go into specifying the operation sets, as well as the manner in which they are mixed constitute a *synthesis* process.

Regarding this process, some implementations found in the literature and industry usually employ simple memory- or cpu-intensive sets of operations in a predefined order to obtain a presumably realistic workload [10, 12]. Other, more sophisticated methods utilize randomness to shuffle the order in which the operations are executed. Building upon this idea, the operations are sometimes executed as prescribed by a distribution, in what is known as *statistical simulation* [9, 40].

Importantly, synthetic benchmarks can also be used similarly to fault-injection frameworks [41, 42]. Since the focus is on replicating real-world workloads, we can conceive of real-world programs that experience performance (or other non-functional) faults. As such, the synthesis methods described above can be utilized to replicate such hypothetical, fault-ridden programs, ensuring that a target component or system will always experience some level of saturation or significant increase in latency.

3 APPROACH

In this chapter, we present the approach followed in addressing the issues outlined in Chapter 1. In the first part, we describe the solution at an abstract level, discussing notions of non-functional properties, gadgets and benchmarks. In the second part, we formalize and discuss some of the notions in our approach that relate to the feasibility of our goals, and organize the main intuitions through concepts like *Impact Mappings* and *Ground Truths*.

3.1 OVERVIEW

The overarching goal of this thesis is to provide a structured way of producing synthetic benchmarks with varied, stable and transparent behaviors. Oftentimes, the term *behavior* is understood as a qualitative extension of functionality, in the sense that it denotes whether some program's functionality lives up to certain expectations of quality. A typical example for this would be performance, e.g. we could wonder whether a program is fast in completing a task, or if it uses little resources in doing so. As such, we take some liberty in using the term *behavior* interchangeably with *non-functional property*. However, it should be noted that non-functional properties comprise more than just performance considerations. In fact, researchers often take an interest in other non-functional aspects such as reliability or availability, to name a few [35–38], which is why we would want our approach to not be confined strictly to performance properties.

On the other hand, to design synthetic benchmarks that have stable behaviors, we need to take a resource-focused approach whereby we try to make our designs impact specific resources in a computer system. Notably, we take an interest in resources that relate to the Northbridge (cf. Chapter 2), because of their direct relevance to software performance considerations and research value [19, 28]. Moreover, considering the overall complexity of modern architectures, it is important for us to focus on one resource at a time. This would allow us to tune and measure program behaviors more easily. However, taking such an approach might appear disconnected from the ambition of addressing many types of non-functional aspects. To that end, we posit that there is a natural congruence between non-functional ideas and resource-specific considerations.

In Chapter 2, we touched upon the notion of resource-specific metrics, which directly relate to a program's behavior with regard to a specific architectural resource in a computer system. We also touched upon the notion of (runtime) Non-Functional Properties (NFPs) and how they typically need to be made measurable and quantifiable. Moreover, we discussed how, based on the utilities provided by Performance-Monitoring Units (PMUs), we are able to obtain measurements for resource-specific metrics. Once we have measurements in place, we can rely on open standards such as *Systems and software Quality Requirements and Evaluation (SQuaRE)* [39] to determine what NFPs can be of interest and how we can relate our measurements to them. Thus, a tentative model that bridges NFPs and metrics is shown in Figure 3.1.

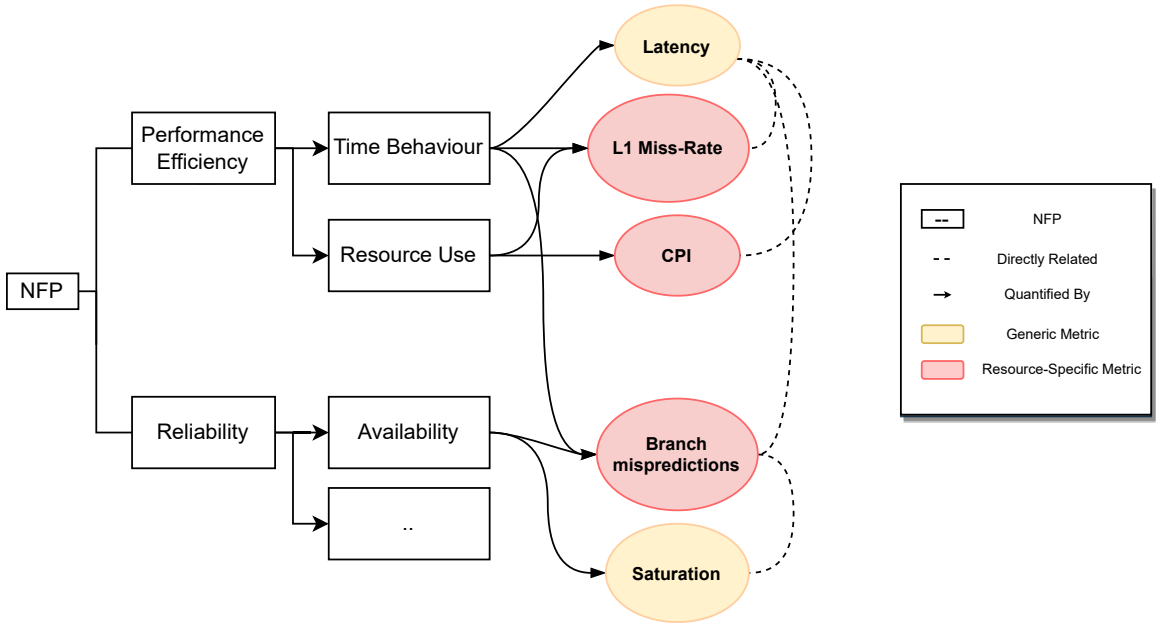

Figure 3.1: NFPs and types of metrics. NFPs are broken down into sub-attributes. The sub-attributes are then quantified using generic or resource-specific metrics (cf. Chapter 2)

Based on the figure, we can consider a case in which the overall performance[1] of a program can be broken down into two conceptual properties: Time Behaviour and Resource Use. The former refers to the extent to which the processing time of a program affects its quality, whereas the latter refers to the extent to which the usage of certain resources meets predefined expectations. To make it more specific, Time Behaviour can be quantified using metrics such as Latency, which can be measured without focusing on any resource in particular. Furthermore, we can dive deeper and obtain metrics about specific resources such as the caches. For instance, we can use the Level-1 Miss-Rate metric; as mentioned in Chapter 2, cache-misses cause delays in terms of the processing of data, which contributes to latencies and implies that the program will require more time to complete its execution, hence the direct connection to the Latency metric in the figure, but also to the Time Behaviour property. An analogous idea applies to the Cycles per Instruction (CPI) metric, since it simply quantifies the efficiency of the Central Processing Unit (CPU) and directly indicates both Time Behaviour and Resource Use. In a similar manner, a metric like Branch Mispredictions can indicate and quantify the property of Availability, which can be understood as the ability of a program to respond to users while it is executing. This is the case because mispredictions can lead to stalls in the CPU pipeline which, in extreme cases, can cause the process to appear non-responsive. As such, the program might not be able to immediately start executing new instructions, thus reducing its availability.

The example we just discussed serves to bridge the ideas of focusing on specific resources, while still being able to speak in terms of high-level properties. While the precise ways metrics and NFPs interrelate are not defined in a strict or comprehensive way in the aforementioned standard, the guiding principle is that the more metrics we can use, the more information we can obtain regarding NFPs. In any case, our understanding is that there are

1 In the standard, performance is typically referred to as "Performance Efficiency".

Metric	Description	Component	Tools	Scope
Level-1 Miss-Rate	The rate in which misses occur in the first-level data-cache, obtained by the ratio of misses and total cache references.	Memory Subsystem, CPU-core	perf	process, system-wide
Branch Misprediction Rate	The ratio of branch mispredictions and overall branching instructions.	Branch Prediction Unit, CPU-core Frontend	perf, bpftrace[28]	process, core, system-wide
Cycles per Instruction	The number of CPU cycles spent in executing an instruction, on average.	CPU-core Backend	perf	process, system-wide

Table 3.1: Examples of resource-specific metrics, calculated on the basis of specific counters. See Chapter 4 for complete lists of such metrics.

no limitations as to how one can relate metrics and NFPs. Based on this idea, for the rest of this thesis we do not make explicit use of NFPs, and instead focus only on metrics without risking any loss of generality.

In Table 3.1 we list several examples of resource-specific metrics, together with the architectural component they relate to, using the notions presented in Chapter 2. In addition, we indicate the scope of the metric, i.e. the level at which the metric is informative. Notably, we can attribute measured values to one specific process (narrow scope), as opposed to all the running processes in a system (wide scope). For instance, the Level-1 Miss-Rate metric corresponds to Level-1 caches, and can be measured for any single process with the aim of evaluating the cache-relevant properties of that process. This is made possible by tools like *perf*, which can reliably obtain counter values from the PMU, for any single process. The aggregation of these counters' values is handled by us once they are available, and the metrics that we obtain by aggregating these counters' values are also process-specific. Overall, we mostly rely on *perf* and augmented versions of it that enable us to make use of various counters by exposing the Model-Specific Register (MSR) interface to the user-level. For debugging purposes, we also rely on profiling (cf. Chapter 2).

Working in a bottom-up manner, and based on the NFP-metrics congruence, we then look towards high-level programming techniques that allow us to build programs that achieve stable values for resource-specific metrics, and consequently stable behaviors. That requires a mix of intensive workloads and misplaced/reversed optimizations, implemented in a mid-level language such as C++. Once we have identified such techniques, we implement and structure them using Feature-Oriented Software Development (FOSD) notions such as features. That provides us with configurable components to which we refer as *gadgets*, and an explicit view of these techniques in the implementation. Gadgets then provide us with the first step towards constructing actual synthetic benchmarks.

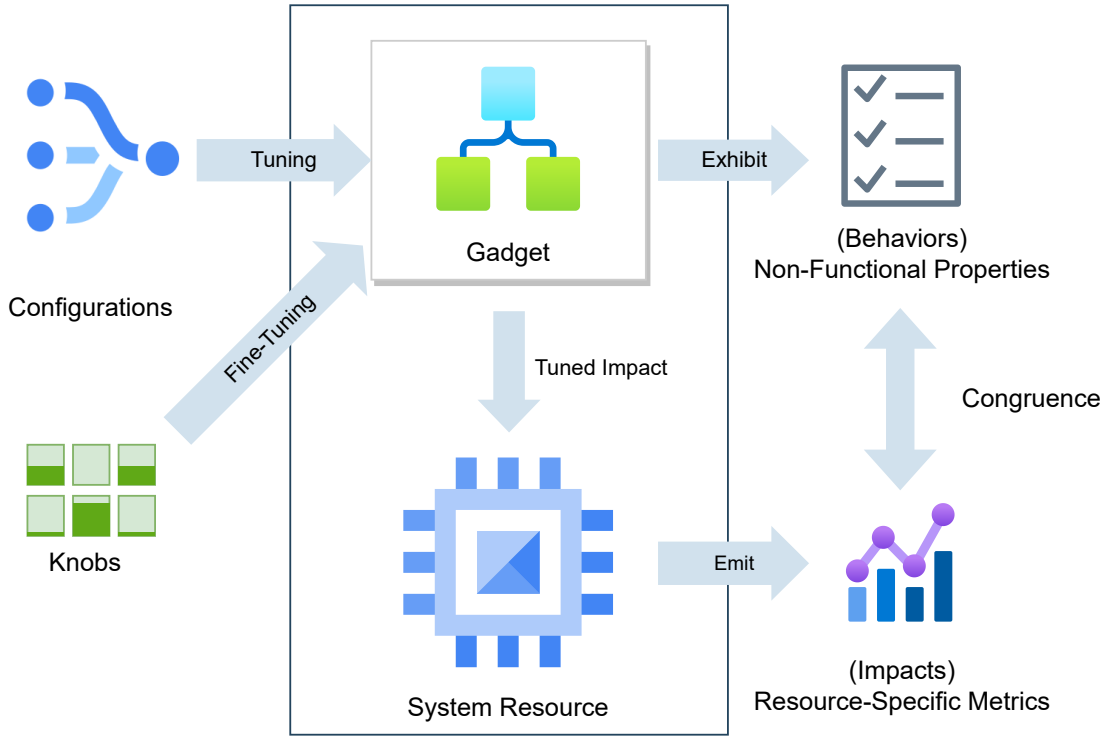

Figure 3.2: The Gadget Concept.

The full concept of a gadget is shown in Figure 3.2. On the right side of the figure, we highlight the congruence of NFPs and resource-specific metrics discussed previously. On the left side we depict how the impact of a gadget can be tuned and varied on the basis of its configurability. In this context, we use the term *tuning* to indicate the ability to obtain different behavioral profiles for a gadget. However, configurability at the level of features might not suffice. Indeed, any two variants of a gadget will target the same system resource, yet will typically have considerably distinct behavioral profiles w.r.t. that resource. While this is a desired aspect of our approach, it might also be useful to enable some slight variations in the performance profile of each variant. Therefore, we also enable finer-grained tuning by relying on other configurable parameters (knobs) that do not constitute features, but rather simple workload scalers.

To facilitate a more structured workflow for researchers, as well as to enable code reuse, we organize the gadgets in Software Product Lines (SPLs), whereby each valid configuration of a gadget corresponds to a variant with a distinct behavioral profile. In other words, each gadget variant conceptually represents a standalone product, which can be used independently of other gadgets. The set of these variants represents a relatively small SPL, and the artifacts that are reused are also few. While we could opt to add a large number of features to a single gadget, we believe that would defeat the purpose of focusing on specific system resources.

Building on top of this SPL idea for the gadgets, we then enable an approach that allows the researcher to combine multiple gadget variants into one execution sequence. We make this possible by complementing the SPLs with automated binding and derivation flows, which also help in terms of maintainability. Then, provided that gadget variants can be

derived automatically, we take inspiration from *fault injection* notions to enable their use in a larger setting [41, 42].

More specifically, we make it possible to *inject* gadget variants in the static flow of existing programs that we also design. While it is possible to inject the variants in arbitrary codebases, we intend to pursue this idea in future work (cf. Chapter 7). Instead, we make use of miniature, *template* systems for which we can ensure that the gadget variants can be injected, and in which we manually specify where exactly the injections should happen. These systems are inherently configurable and feature-oriented, with any feature or (structural) feature-interaction therein potentially able to accommodate multiple injections of gadget variants. Importantly, these systems are assumed to present a minor workload of their own, such that the template itself does not convolute the performance of any injected gadget variant. As such, we are able to perform workload synthesis, whereby the mixing of workloads is determined by the static execution flow of the template system, and the workloads are presented by the injected gadget variants.

Considering how the template systems are inherently configurable, it is easy to organize them in SPLs of their own, after injecting gadget variants in their static flow. That means that we could organize our overall approach as a two-level SPL concept, based on which we maintain some link between variants of the template system to the variants of the gadgets. However, that poses some additional challenges. For instance, we might want to obtain a variant of a template system, and therefore we make a feature-selection. Then, to bind this selection we would need to know how the features of the template system map to gadget variants (or even more specifically, gadget features), so that those gadget variants are generated, injected and initialized in the correct locations in the template system's code, before the feature-selection is actually enacted for the template system itself. Taking into account the possibility that the implementation of features of the template system can be scattered across code blocks and compilation units, this becomes challenging. To address this, we opt to decouple the workings of the gadget SPLs from those of the template systems. The injection-based approach naturally enables this by allowing us make decisions regarding the template SPL separately (asynchronously) from the decisions regarding the gadget SPLs. For instance, at one point in time we can derive various gadget variants, and at a later point we can think about where and how to inject these variants in a template, and what variant of the template to derive after injection.

Building upon the aforementioned ideas, we are ultimately able to organize our approach into a unified concept for deriving synthetic benchmarks, depicted in Figure 3.3. In the figure, we can distinguish two main processes: 1) a *derivation* process through which we obtain different variants of a gadget by making use of its feature-oriented implementation, and 2) a *weaving* process through which we make use of the gadget variants to obtain a synthetic benchmark, based on the injection idea discussed previously. Practically, the steps we can follow based on this concept are summarized as follows:

1. Firstly, we define a specification regarding gadget use and benchmark creation. This artifact specifies the gadget variants and template systems we want to use, as well as how we want to inject the gadgets variants in the template systems. As such, it is relevant to both processes depicted in the figure.

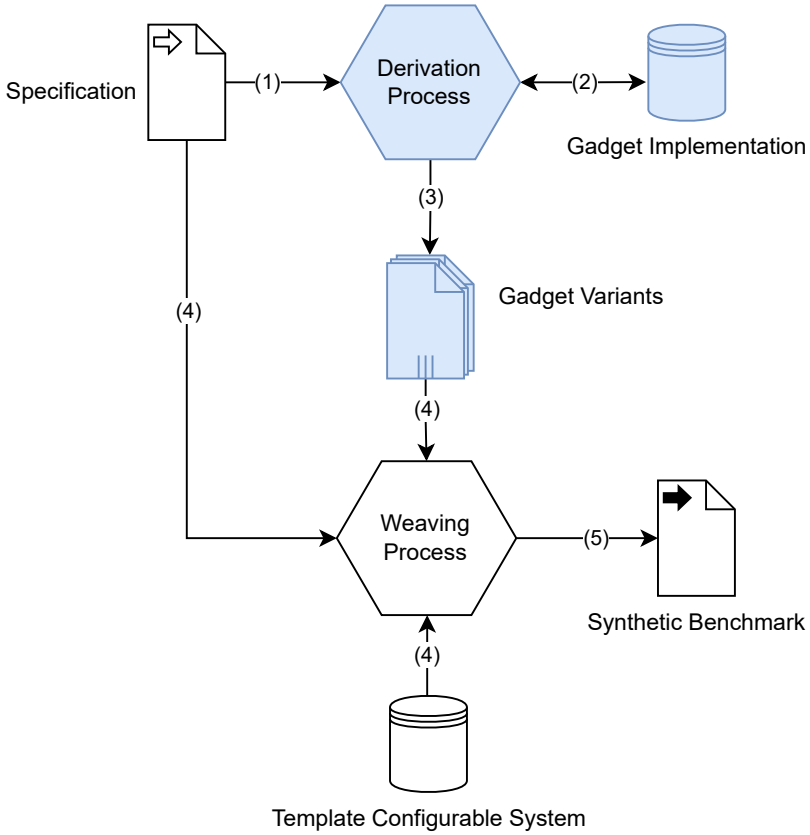

Figure 3.3: Synthesis Concept. Parts pertaining to the gadget SPLs are highlighted in blue.

2. Secondly, we bind feature-selections for specific gadgets and derive various gadget variants through the derivation process.

3. Thirdly, the derivation process concludes with the outputting of multiple gadget variants' artifacts, while handling any issues that might hinder the next step.

4. At the fourth step, the weaving process commences, contingent on the specification provided in the first step. The already generated gadget variants are used as input artifacts. In addition, a template (configurable) system will be provisioned to the process. Using these inputs, the gadget variants are combined and injected in the static flow of the template system.

5. Lastly, after injecting and combining gadget variants in the template system, the weaving process concludes by binding a feature-selection for the template system, yielding a *synthetic benchmark*.

Crucially, the resulting synthetic benchmark will have largely stable and transparent behaviors. This can be seen by the fact that the gadgets are designed in such a way that they exhibit stable and transparent behaviors, and they can be weaved under any of the existing features or (structural) feature-interactions of a given template system, which themselves are known and transparent. Based on this idea, and the assumption that the template system presents only minor workloads of its own, the overall behaviors of the template system

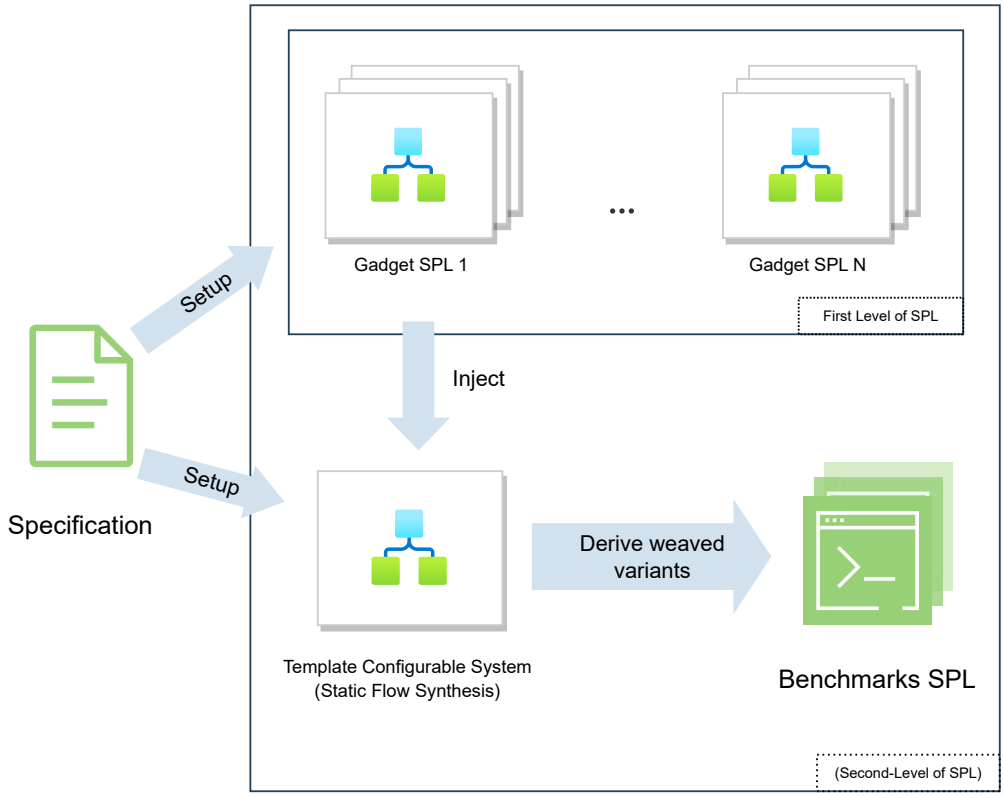

Figure 3.4: Two-Level SPL concept in full display.

with injected gadget variants will also be stable and transparent. In other words, we can reliably know the specific, gadget-defined, and thus stable behavior of any of the features or configurations of a given template, after weaving. That being said, potential exceptions might exist when the features of a given template system interact in unexpected ways, which brings about some instability. Even if that is the case, the researchers can still benefit from the knowledge that a certain way mixing of gadget variants' workloads presents an interesting behavior, and that this mixing is transparent.

To conclude this part of the discussion, we provide a closer view of the two levels of SPLs that result from our approach, in Figure 3.4. At the top, we notice a set of gadget product-lines, which collectively represent the first level of our two-level solution. Then, in the bottom we have the template system, which in itself is configurable, as pointed out earlier. A crucial idea here is that for the template system we only derive a variant after having weaved some gadget variants in fixed locations of the template's source code. In other words, a template variant constitutes a benchmark only if such a weaving is performed. In addition, considering that the locations where the gadget variants are injected are statically defined, we say that the workload of the resulting benchmark is synthesized by means of the static execution flow of the template. Moreover, since the template system itself could (in principle) be organized in a product-line, we are able to obtain a family of benchmarks by executing the weaving process multiple times for different variants of the template system.

3.2 FORMALIZATION

Having discussed the overall approach in some detail, there are certain aspects that are missing or need to be made more precise in order to gain a better understanding of this work. Therefore, in this section we provide formal notation and descriptions of the main ideas surrounding gadgets and benchmarks, augmented with simple examples.

3.2.1 *The Gadget*

To begin our formalization, we take a deeper look into gadgets. We previously introduced the overall concept of a gadget, which is the main building block that we use to construct synthetic benchmarks. In this section, we make that concept more precise and formal. To that end, we introduce definitions regarding knobs, valid configurations, and resource-specific metrics, before precisely describing what gadgets are.

To begin with, we introduced the concept of configurations and valid configurations in Chapter 2, though we need to make it more precise by providing a clear definition as follows.

Definition 1. (*Valid Configuration*)

> Let v be any selection of features for a given configurable software system that abides by the feature-model of said system. Then, we say v is a *valid configuration* of that system. We denote a set of valid configurations as $\mathcal{V}$ s.t. $v \in \mathcal{V}$.

Normally, the set of all configurations for a given configurable program is denoted as $\mathcal{C}$, therefore $\mathcal{V} \subseteq \mathcal{C}$. Note that from this point onward, we take some liberty in referring to valid configurations as *variants*, considering how each valid configuration effectively enables us to obtain a variant of a system (cf. Chapter 2).

Besides the feature-based configurability of gadgets, we also enable some fine-tuning through configurable parameters called knobs, as explained in the previous section. To that end, we clearly specify what a knob is in the following.

Definition 2. (*Knob*)

> Let k be any configurable parameter that does not constitute a feature, but which can be used to scale the workload that a configurable program or system applies to the resources of a computer system. Then, we say k is a *knob*. We denote a set of knobs as $\mathcal{K}$ s.t. $k \in \mathcal{K}$.

The last ingredient we need to clearly specify what a gadget is, is that of resource-specific metrics. The notion was introduced in Chapter 2, and is formally defined as follows.

Definition 3. (*Resource-Specific Metric*)

> Let μ be any statistic of interest that is obtained through low-level performance counters for the purpose of quantifying the behavior of a configurable program or system w.r.t. an architectural resource of a computer system. Then, we say μ is a *resource-specific metric*. We denote a set of resource-specific metrics as $\mathcal{M}$ s.t. $\mu \in \mathcal{M}$.

Based on the above definitions, we are now able to precisely define gadgets, presented as follows.

Definition 4. (*Gadget*)

> Let g be a configurable program that has pronounced and stable impacts on specific hardware resources, and for which we can always define the following properties:
>
> - A set $\mathcal{V}_g = \{v \mid v \text{ is a valid configuration of } g\}$
> - A set $\mathcal{K}_g = \{k \mid k \text{ is a knob of } g\}$.
> - A set $\mathcal{M}_g = \{\mu \mid \mu \text{ is a resource-specific metric that quantifies the behavior of } g\}$.
>
> Then, we say that g is a *gadget*.
>
> Moreover, we say that $g \in \mathcal{G}$ where $\mathcal{G}$ is the set of all gadgets. To distinguish between properties of different gadgets, we use subscripts $\mathcal{K}_g$, $\mathcal{V}_g$ and $\mathcal{M}_g$, where g denotes the gadget to which any of these sets correspond.

From the definition, we can discern several characteristics of gadgets. To begin with, each gadget is built using FOSD practices, thus having a well-defined feature-model. Its inherent configurability makes the gadget useful in obtaining a variety of behavioral profiles. In addition, a gadget does constitute a standalone software system (program) in the technical sense, though it is not used as such in our approach. Moreover, a gadget can be understood, at an abstract level, as including not only the implementation but also a well-defined set of metrics specific to some architectural resource, so as to make its impacts w.r.t. that resource transparent. This comes as a result of our resource-focused approach.

Before looking into an example of how a gadget can be described more practically, it is important to have some definition in place for the knobs and the values they can be assigned, since they represent a non-trivial aspect of gadgets.

Definition 5. (*Knob Assignment*)

> Let $g \in \mathcal{G}$ be any gadget, and let $k_1, k_2, ..., k_n \in \mathcal{K}_g$ be the set of knobs defined for g. Moreover, let $r_1, r_2, ..., r_n \in \mathcal{R}_0^+$ be the set of positive (including zero) real numbers. Then, a *knob assignment* for g is defined as a mapping of the form:
>
> $$\begin{aligned} &a : \mathcal{K}_g \to \mathcal{R}_0^+ \\ &a = \{(k_1, r_1), (k_2, r_2), ..., (k_n, r_n)\} \end{aligned} \tag{3.1}$$
>
> Moreover, we say that $a \in \mathcal{A}_g$, where $\mathcal{A}_g$ is the set of all such assignments for the knobs of gadget g.

It should be noted that a knob can always be assigned a value, but for some variants that assignment might have no effect. This largely depends on the gadgets' features and how they are implemented, though we try to make knobs as general-purpose as possible. We discuss this further in Chapter 4.

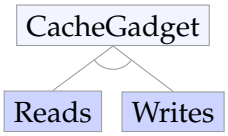

Figure 3.5: Simplified feature-model for the Cache Gadget. The model allows for either Reads or Writes to be enabled.

Example. (*A simple gadget*)
To better understand gadgets, we make use of a simplified description of a gadget we have built, called the Cache Gadget, denoted as cg. The gadget is designed to achieve a high number of cache-misses across all levels of caching. In this simplified example, the gadget's workload conceptually consists of either reads or writes, something which we capture in the feature-model shown in Figure 3.5. Using this feature-model, we obtain the set of valid configurations:

$$\mathcal{V}_{cg} = \{(CacheGadget, Reads), (CacheGadget, Writes)\}$$

For brevity, we will refer to the respective configurations using indexed labels, v_1, v_2 and so on. That is:

$$v_1 = (CacheGadget, Reads),$$
$$v_2 = (CacheGadget, Writes)$$

where $v_1, v_2 \in \mathcal{V}_{cg}$.

We can then define a set of metrics that can capture the impact of the gadget adequately:

$$\mathcal{M}_{cg} = \{MissRate, CPI\}$$

Similarly, we can define a set of knobs that allow us to scale the impact of the gadget:

$$\mathcal{K}_{cg} = \{WorkloadAmplifier\}$$

In this example, the $WorkloadAmplifier$ knob is conceptualized as a variable which serves the purpose of increasing the amount of reads or writes the gadget performs. For instance, a value of 0.4 assigned to the knob would indicate a 40% increase in the workload, on top of some default amount. As such, we posit that the $WorkloadAmplifier$ variable takes ratio values, i.e.:

$$dom(WorkloadAmplifier) = [0, 1]$$

Note that this amplification is presumably useful for both valid configurations v_1 and v_2. That may or may not be the case for all knobs, since the usefulness of a knob largely depends on the actual feature implementations (cf. Chapter 4). In any case, we then have the following assignment $a \in \mathcal{A}_g$:

$$a = \{(WorkloadAmplifier, 0.4)\}$$

3.2.2 *Impact Mappings*

Having established a definition for gadgets, we need to address its important properties of having stable and transparent behaviors. As mentioned in the previous section, gadgets should provide us with stable behaviors so that they can be reliably used in our overall process of constructing synthetic benchmarks. As such, the synthetic benchmarks that are build from them can also have stable and consequently transparent behaviors, which in turn would help us address the practical need for configurable software systems the behavior of which is provided out of the box, i.e. without having to perform analysis on them (cf. Chapter 1).

Therefore, to quantify and document the behaviors of gadgets, we formally describe the relations between knobs, valid configurations and metrics. This way we can have a structured description of a gadget's behavior, alongside its implementation, which would allow us to clearly see and verify the stability of a gadget's behaviors and help us make them transparent as intended.

To that end, we posit that the variants of a gadget have a specific relation to the metrics that are associated with said gadget. This is a result of our resource-focused approach whereby each gadget variant targets specific metrics, in the sense that the workload it applies onto system resources is such that the values of these metrics are relatively pronounced, as well as stable. This relation is defined as follows.

Definition 6. (*Targeting Relation*)

Let $g \in \mathcal{G}$ be any gadget. Then, the targeting relation is defined for gadget g as:

$$\begin{aligned} \mathcal{V}_g \triangleright \mathcal{M}_g = \{ & \\ & (v, \mu) \in \mathcal{V}_g \times \mathcal{M}_g \mid \text{variant corresponding to } v \text{ obtains values} \\ & \text{w.r.t. } \mu \text{ that are pronounced and highly stable.} \\ & \} \end{aligned} \tag{3.2}$$

The relation is denoted with the operator $\triangleright$.

Based on the definition above, we can say that for a given variant $v \in \mathcal{V}_g$, not all metrics in $\mathcal{M}_g$ might be useful, which is the reason why we define the $\triangleright$-relation instead of taking a cartesian product between $\mathcal{V}_g$ and $\mathcal{M}_g$. In fact, a cartesian product would combine each $v \in \mathcal{V}_g$ with each $\mu \in \mathcal{M}_g$, whereas sometimes we want to combine v with only some of those metrics. Moreover, if some variant does not target a metric, the values that the metric can take can vary freely.

In any case, it should be noted that what constitutes a *pronounced* value for a metric largely depends on the specific resource that metric relates to and the conventional knowledge surrounding it. We follow domain knowledge in that regard, e.g. miss-rates beyond 20% on the Level-3 cache can be considered detrimental to performance and other non-functional properties [19, 43].

In addition to the $\triangleright$-relation, we note that the impact that any gadget variant has on its targeted metrics can be slightly varied through the knobs. These variations are limited by

design, and serve only for fine-tuning purposes. However, they still play a role in the overall behaviors of a gadget and need to be part of the description of these behaviors.

Then, based on the ideas discussed thus far we can define a formal description of a gadget's behaviors in what we call an Impact-Mapping (IM).

Definition 7. (*Impact Mapping*)

> Let $g \in \mathcal{G}$ be any gadget, and let $\mathcal{R}_0^+$ be the set of positive (including zero) real numbers and $\mathcal{N}$ be the set of positive integers. Then, an *impact-mapping* of gadget g is defined as follows:
>
> $$\mathcal{IM}_g : (\mathcal{V}_g \triangleright \mathcal{M}_g) \times \mathcal{A}_g \rightarrow \mathcal{R}_0^{+^n} \tag{3.3}$$
>
> where $n \in \mathcal{N}$, and $\mathcal{R}_0^{+^n}$ is the n-fold cartesian product of $\mathcal{R}_0^+$.

Based on the above definition, we firstly notice how the IM combines the variants with the metrics that they target through the $\triangleright$-relation. Moreover, we have already mentioned that the knobs can slightly modify the behavior of a variant, which is why we need to include $\mathcal{A}_g$ in the IM; specifically, for the same (v, μ) pair, we can use different assignments $a_1, a_2, ..., a_m \in \mathcal{A}_g$ for the knobs, which will result in slightly distinct values for the metric μ in that pair. Overall, for each tuple of the form $((v, \mu), a)$ we can obtain a group of measurements for the metric μ under consideration. The number of values n in each such group is defined based on statistical considerations (cf. Chapter 5). The benefit of obtaining multiple values is that we can effectively construct frequency distributions for each tuple, which allows us to better verify the stability of the behaviors resulting from variants and knob assignments. Crucially, we intend to make these behaviors transparent, and the notion of an IM directly helps in this regard.

Before looking at an example, we should note that it might be useful to omit the knobs from considerations regarding gadgets' behavioral profiles, due to their limited effects in altering the behavior of gadget variants. In such cases, we can simply fix an assignment for the knobs and simplify the overall notation. This provides us with a *relaxed* notion of an IM, which we define as follows.

Definition 8. (*Relaxed Impact Mapping*)

> Let $g \in \mathcal{G}$ be any gadget, $\mathcal{R}_0^+$ be the set of positive (including zero) real numbers and $\mathcal{N}$ be the set of positive integers. Moreover, let $\mathcal{B} \subseteq \mathcal{A}_g$ be the set of those knob assignments for which we get the largest impact possible for each variant $v \in \mathcal{V}_g$, w.r.t. the metrics each such variant targets. Then, a *relaxed impact-mapping* of gadget g is defined as follows:
>
> $$\mathcal{IM}_g^{\mathcal{B}} : \mathcal{V}_g \triangleright \mathcal{M}_g \rightarrow \mathcal{R}_0^{+^n} \tag{3.4}$$
>
> where $n \in \mathcal{N}$, and $\mathcal{R}_0^{+^n}$ is the n-fold cartesian product of $\mathcal{R}_0^+$.

For any variant of g, $\mathcal{B}$ contains a knob assignment for which that variant should obtain maximal impacts, w.r.t. the metrics that it targets. Naturally, $IM_g^{\mathcal{B}} \subseteq IM_g$. The relaxed form

of IM is thus an approximation of the actual IM, and helps us look at only the highest possible impact of each variant. This relaxation will prove helpful to us during evaluation, in Chapter 5.

Example. (*An Impact Mapping*)
To construct an example for the IM, we build upon the previous example. Firstly, assume we use an assignment a for the knobs as in the previous case, and the same variants v_1 and v_2 of the Cache Gadget. Then, we can conceive of a scenario where we perform measurements for each of the two variants, while using the same knob assignment a as in the previous example. For each case, we can obtain two measurements ($n = 2$). However, we first need to know what metrics each of the variants targets. Naturally, the $\triangleright$-relation tells us exactly this, as it is defined alongside the gadget's design and implementation. To that end, we can have the following:

$$\begin{aligned}\mathcal{V}_{cg} \triangleright \mathcal{M}_{cg} = \{ & \\ & (v_1, CPI), \\ & (v_2, CPI), \\ & (v_2, MissRate) \\ \}&\end{aligned}$$

Based on this, we can construct the following blueprint for an IM:

$$\begin{aligned}\mathcal{IM}_{cg} = \{ & \\ & ((v_1, CPI), a) \rightarrow f^a_{v_1 \triangleright cpi}\ , \\ & ((v_2, CPI), a) \rightarrow f^a_{v_2 \triangleright cpi}\ , \\ & ((v_2, MissRate), a) \rightarrow f^a_{v_2 \triangleright miss\text{-}rate} \\ \}&\end{aligned}$$

Note that we are using $\rightarrow$ instead of simple commas to make the mapping more visible in the pairs of the IM. In addition, we are using the symbol f to label groups of measurements. The subscript in $f_{v_1 \triangleright cpi}$ means that this is a series of measurements for variant v_1, w.r.t. the metric CPI that this variant targets. The superscript in f^a denotes that the knob assignment used for these measurements is a. To get a better idea, we can construe the following hypothetical situation in which we get the following measurements:

$$\begin{aligned} f^a_{v_1 \triangleright cpi} &= (2, 2.5) \\ f^a_{v_2 \triangleright cpi} &= (3, 3.25) \\ f^a_{v_2 \triangleright miss\text{-}rate} &= (0.95, 0.97) \end{aligned}$$

To make the notation clearer, note that the first set of measurements simply indicates that the CPI values for variant v_1, with knob assignment a are 2 and 2.5, across two independent

measurements conducted as part of some experiment. Based on this, and the IM blueprint described earlier, we can describe the actual IM:

$$
\begin{aligned}
\mathcal{IM}_{cg} = \{ & \\
& (((CacheGadget, Reads), CPI), (WorkloadAmplifier, 0.4)) \rightarrow \\
& (2, 2.5), \\
& (((CacheGadget, Writes), CPI), (WorkloadAmplifier, 0.4)) \rightarrow \\
& (3, 3.25), \\
& (((CacheGadget, Writes), MissRate), (WorkloadAmplifier, 0.4)) \rightarrow \\
& (0.95, 0.97), \\
\} &
\end{aligned}
$$

From this example IM we can clearly see that different variants lead to different impacts w.r.t. the same metric, indicating the benefit of configurability; indeed, using the same code artifacts, we can obtain different behaviors. For simplicity, we did not include the case when we use different assignments for the knobs; in a case with two possible knob assignments, we would have to include twice as many entries in the IM, since each variant would have to be considered separately for each knob assignment, and therefore we would need to obtain distinct measurements for each case.

3.2.3 *Ground Truth*

The notions introduced thus far serve to organize our approach in designing and realizing the gadget idea. The introduction of the IM helps structure our intuitions regarding our resource-specific approach, especially regarding the verification of a gadget's correctness and making of its behaviors transparent. Nevertheless, the IM provides only the first step in the quest for transparency. To see why that is the case, we can consider that any IM we obtain for our gadgets is likely to vary significantly across environments, which makes it difficult for us to say that a gadget will maintain its behaviors (i.e. IM) in any environment it is used. In other words, the IM helps us make gadget behaviors transparent only if these behaviors are guaranteed to be stable across environments. However, microarchitectural differences across environments can make for large differences in the impact that a certain workload can attain [9].

Before addressing this issue, we should specify what is meant by the term *environment* more formally.

Definition 9. (*Environment*)

> Let e be any computer system that can be uniquely identified on the basis of its microarchitecture. Then, we say e is an *environment*. We denote the set of such environments as $\mathcal{E}$ s.t. $e \in \mathcal{E}$.

Having this definition in place, we posit that the issue of instability across environments can be tackled by experimentally obtaining IMs in multiple distinct systems. Naturally, this need only be done once for each environment if the gadget is known to be consistent in its

behaviors in that environment. However, in order to make our gadgets as useful as possible, and to lift the burden of manually specifying IMs, we would ideally like to have some way of indicating that the IMs we obtain for certain environments can naturally generalize to other environments. This depends on how reproducible the behaviors of our gadgets are, which is something we evaluate in Chapter 5.

In any case, we can benefit from a generalization of the notion of IM, which accounts for the variety in behaviors that is due to environments. To that end, we provide the following definition.

Definition 10. (*Ground Truth*)

> Let $g \in \mathcal{G}$ be any gadget, $\mathcal{R}_0^+$ be the set of positive (including zero) real numbers, and $\mathcal{N}$ be the set of positive integers. Moreover, let $\mathcal{E}$ be the set of known environments. Then, the *ground-truth* for gadget g is defined as:
>
> $$\mathcal{GT}_g : \mathcal{E} \times \underbrace{((\mathcal{V}_g \triangleright \mathcal{M}_g) \times \mathcal{A}_g) \rightarrow \mathcal{R}_0^{+^n}}_{\text{Impact Mapping}} \tag{3.5}$$
>
> where $n \in \mathcal{N}$, and $\mathcal{R}_0^{+^n}$ is the n-fold cartesian product of $\mathcal{R}_0^+$.

What we observe in the above definition is that we are still using all the constructs we used in defining IM, but now we are accounting for the effects of environments to the measured values, therefore generalizing the notion of an IM as intended. Moreover, given the similarity to IM, we can also relax the notion of Ground Truth (GT) by fixing an assignment for the knobs as before.

Definition 11. (*Relaxed Ground Truth*)

> Let $g \in \mathcal{G}$ be any gadget, $\mathcal{R}_0^+$ be the set of positive (including zero) real numbers, and $\mathcal{N}$ be the set of positive integers. Also, let $\mathcal{E}$ be the set of known environments. Moreover, let $\mathcal{B} \subseteq \mathcal{A}_g$ be the set of those knob assignments for which we get the highest impact possible for each variant $v \in \mathcal{V}_g$, w.r.t. the metrics each such variant targets. Then, the *relaxed ground-truth* for gadget g is defined as:
>
> $$\mathcal{GT}_g^{\mathcal{B}} : \mathcal{E} \times \underbrace{(\mathcal{V}_g \triangleright \mathcal{M}_g) \rightarrow \mathcal{R}_0^{+^n}}_{\text{Relaxed Impact Mapping}} \tag{3.6}$$
>
> where $n \in \mathcal{N}$, and $\mathcal{R}_0^{+^n}$ is the n-fold cartesian product of $\mathcal{R}_0^+$.

Similarly to the case of IM, the relaxed GT is an approximation of the actual GT in which the knob assignments are fixed and implied for each variant.

As a last consideration regarding IMs and GTs we should note that, during implementation, we rely on a reference environment to build and test the gadgets. As such, we make a slight distinction between IMs that can be obtained in arbitrary environments, and the IM that can be obtained in the reference environment. Specifically, we refer to an IM obtained in the reference environment for a gadget as the Reference Impact-Mapping (IM_{ref}) of that gadget[2]. Based on the IM_{ref}, we are able to see whether a gadget's behaviors are consistent across

2 We introduce and use the reference environment in Chapter 5, during evaluation.

multiple executions in the same environment, before trying to evaluate the overall stability of gadgets across different environments.

3.2.4 *Benchmarks*

In Section 3.1, we introduced the overall approach that allows us to construct synthetic benchmarks on the basis of gadgets. Now that we have some formal definitions in place for gadgets, we aim to make the discussion around the synthetic benchmarks more precise.

As a first step, we need to specify how our approach handles the injection of gadget variants in a given template system. This requires us to take into consideration the configurable nature of template systems; in particular, we need to consider the fact that the features of a template configurable system can be scattered across its code artifacts, which means that it is difficult to associate and track any injected gadget variants w.r.t. template features. To address this issue, we introduce some indirection based on which we map in-code locations, instead of features of the template system, to gadget variants. This allows us to inject gadget variants in any location, without necessarily having an explicit mapping between features of the template system to gadget variants. Ensuring that the injections are performed under specific features (or even feature-interactions) of the template system is then part of the implementation process of that system itself.

Before we address the ideas of injections and benchmarks, we provide the following definitions for template systems and locations in order to avoid any misconceptions.

Definition 12. (*Template System*)

> We define a *template system* as a configurable software system, in the execution flow of which gadget variants can be weaved.

Definition 13. (*Location*)

> Let s be some template system, and let l be any specific line of code in a code file of s, which can be used to inject a gadget variant within the execution flow of that system. Then, we say l is a *location* of s. We denote the set of such locations for system s as $\mathcal{L}_s$ s.t. $l \in \mathcal{L}_s$.

Based on the definitions of template systems and locations, we can now define a mapping between in-code locations of the template system, and gadget variants belonging to any gadget, in what we call Injective System-Gadget Mapping (ISG). The definition is as follows.

Definition 14. (*Injective System-Gadget Mapping*)

> Let s denote any template system, and let $\mathcal{G}$ denote the set of all gadgets. Also, let $\mathcal{L}_s$ describe a set of locations in the implementation of s. Then, we define an *injective system-gadget mapping* for system s as:
>
> $$\mathcal{ISG}_s : \mathcal{L}_s \rightarrow \bigcup_{g \in \mathcal{G}} \mathcal{V}_g \tag{3.7}$$

Based on the above definition, we can inject variants of any gadget into any of the in-code locations of a given template system. The locations can belong to different features or feature-interactions of the template system, based on which we can obtain a group of variants (of the template) with synthetic behaviors, effectively yielding an SPL of benchmarks. Moreover, for each benchmark, we could redefine its behaviors through the ISG by changing the mapping for existing locations. After modifying the ISG this way we would simply need to re-run the weaving process shown in Figure 3.3 to obtain an updated benchmark. In addition, one thing to note about the above definition is that it does not take knobs into consideration. In short, if for a given variant $v \in \mathcal{V}_g$, for some gadget g, we use different knob assignments, the overall ISG remains the same, though the behavior of the resulting synthetic benchmark can vary slightly if the synthesis is performed anew. Overall, the notion of an ISG helps us in the implementation of our approach, as well as in documenting the way gadget variants are used w.r.t. a template system. In particular, it helps facilitate the variety in the behaviors of benchmarks, which is an important part of what we are trying to achieve (cf. Chapter 1).

Thus far we have relied on a rather generic notion of synthetic benchmarks, based on our discussion in Chapter 2. However, with the notion of gadgets and ISGs in place, we are able to precisely define *benchmarks* in the context of our work.

Definition 15. (*Benchmark*)

> Let s be any template system. Then, we define a *benchmark* as a variant of system s the behavior of which is defined by weaving specific gadget variants in its execution flow by means of some $\mathcal{ISG}_s$.

An important thing to note about the definition above is that part of the benchmark's workload can come as a result of the (template) system's own workload, though we ensure the template systems we use do not have any large impacts on any resources. This way, the behaviors of a benchmark are mostly or entirely synthetic, as a result of our approach.

In addition, one could wonder whether we should describe an IM or a GT at the level of the benchmarks that we obtain on the basis of injected gadget variants. While this is possible, we believe it adds no value to our considerations. As we understand it, if the GTs of gadgets are stable and transparent, and the ISG is specified for a given template system, the resulting benchmark SPL is also stable[3] and transparent enough to the researcher. For instance, if gadget variants are injected in locations that correspond to a specific feature of the template system, knowing how these variants behave provides knowledge w.r.t. that feature in the context of the benchmark that is produced as a result of our approach.

Example. (*Injection of gadget variants in a template system*)

For this part we make use of a "dummy" configurable system, denoted as *ds*, which acts as the template in which we can inject gadget variants. As such, we can have a set of locations such as the following:

3 Even in cases of exceptions, instability at the level of benchmarks makes for interesting behaviors, as mentioned in Section 3.1.

$$\mathcal{L}_{ds} = \{\overbrace{\text{"/path/to/dummy/fileA:32"}}^{\text{Line 32 in fileA of the 'dummy' system}}, \text{"/path/to/dummy/fileB:99"}\}$$

Each location can be constructed by concatenating the system path to a file with a specific line number in that file. Having specified some locations, we then make use of the gadget that we introduced in previous examples. We know that this gadget has two variants. The ISG will then look like:

$$\begin{aligned}\mathcal{ISG}_{ds} = \{ & \\ & (\text{"/path/to/dummy/fileA:32"} \rightarrow (\mathit{CacheGadget}, \mathit{Reads})), \\ & (\text{"/path/to/dummy/fileB:99"} \rightarrow (\mathit{CacheGadget}, \mathit{Writes})), \\ \}&\end{aligned}$$

Again, we are using $\rightarrow$ to make the mapped pairs more visible. Also, note that for simplicity we are using the same value for the knobs, in both injections. This does not need to be the case. In addition, for this example we are using only one gadget; in reality we can have multiple gadgets, the variants of which can all be used in the ISG.

4 IMPLEMENTATION

In this chapter, we describe the framework that implements the approach outlined in Chapter 3. We specifically discuss how the two-level Software Product Line (SPL) approach is implemented, while pinpointing the main components and their interfaces. In addition, we describe the features of the gadgets that we have built. Lastly, we discuss some limitations that we have identified during the implementation.

4.1 FRAMEWORK

The approach presented in Chapter 3 is implemented through a framework written in the Python[1] language. We rely on a modular approach whereby we have various components responsible for the gadget SPLs, the template configurable system, and the processes of *derivation* and *weaving* that we depicted in Figure 3.3. In Figure 4.1 we depict a component diagram of the framework and all the relevant interfaces. The main components with which the user interacts directly are organized under the User Subsystem. This subsystem contains the Master component, which can be used to perform a full run of gadget variant derivations and weavings into specified template configurable systems. The Master component is concerned with providing the correct specification to the framework in terms of the gadgets, gadget variants, template configurable systems and corresponding Injective System-Gadget Mappings (ISGs), as well as executing user commands. Similarly, the Experiment component can perform these tasks but only for the purpose of experimentation. This component has additional capabilities through which random ISGs can be obtained and enacted, and measurements can be performed so as to obtain Impact-Mappings (IMs) - we take advantage of this during the evaluation in Chapter 5.

Furthermore, in Figure 4.1 we depict two other subsystems, one which corresponds to the gadget SPLs and the associated derivation process, and the other which corresponds to the weaving process through which we combine and inject variants of the gadgets into existing configurable systems (cf. Chapter 3). To maintain and handle the gadget SPLs we rely on two components, namely the Config Registry and the Gadget Manager. The Config Registry handles the user-provided configurations and makes them available to the Gadget Manager. The Gadget Manager is tasked with the handling of the product lines for a given set of gadgets, as well as with the compilation of gadget variants.

Speaking of user-provided configurations, an example for that of the gadget SPLs is shown in Listing 4.1. Each entry in the listing specifies a variant of the gadget using an explicit identifier, as well as knob assignments. The Gadget Manager is tasked with interpreting these identifiers and ensuring that the correct features are selected when a gadget variant is derived (cf. Section 4.2).

Similarly to the gadget SPLs, the template SPLs and the weaving process are handled through the Config Registry and the Weaver components. The aspects that need to be

1 The framework has been implemented in Python v3.6.

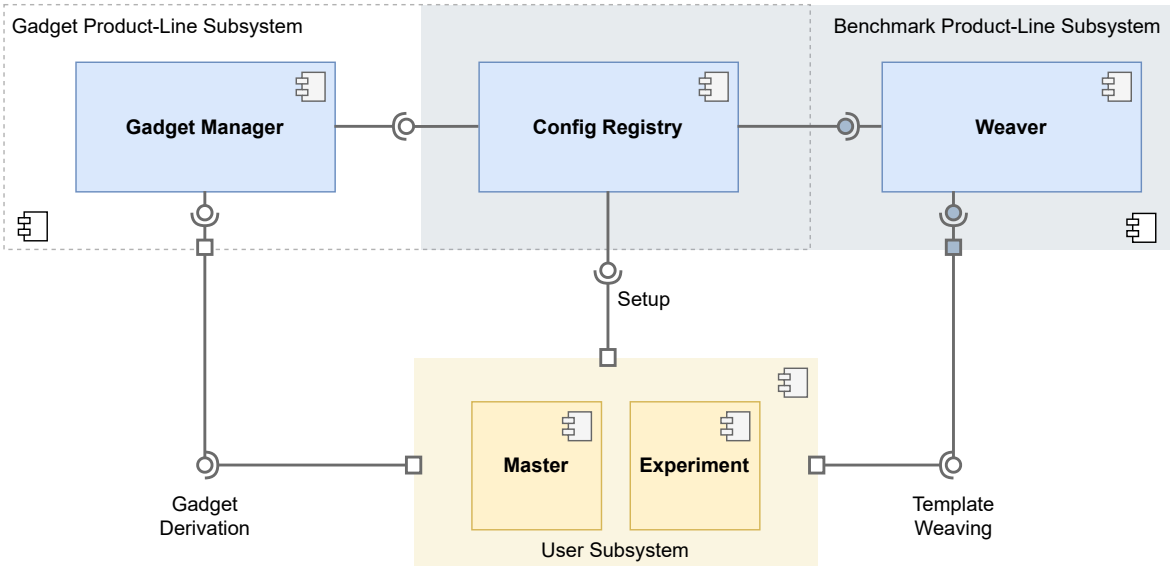

Figure 4.1: Component view of the framework, in UML 2.0 notation [44]. The shaded gray area depicts the subsystem related to the "weaving" process. The area encircled by a dashed line depicts the subsystem related to the gadget SPLs and the corresponding derivation process. The area in yellow depicts the user-facing components. Interfaces are shown using the "lollipop" notation whereby the component connected to the circle provides the interface, and the component connected to the sickle implements that interface. At the level of subsystems, interfaces are unified, shown as small rectangles (ports) on the edge of each subsystem.

```
---
gadgets:
    - name: 'bp_gadget'
        variants:
        - id: c2
            options:
            - <knob_name>: <knob_value>
        - id: c3
        - id: c4
```

Listing 4.1: Sample configuration artifact for the gadget SPLs, where we specify the variant using a named identifier, and an assignment of values to the knobs for each variant. When knob assignments are omitted, a default assignment is used.

specified in this case are two. Firstly, we need to specify a template and a corresponding variant that we want to use, similar to the case the gadgets, except that in this case we only specify one template and one variant. Secondly, we need the ISG to be specified by providing a list of in-code locations for the chosen template and the gadget variants they should map to.

To make our framework more usable, we use numbered identifiers for each in-code location that we call *injection points*, using the general form *IP_<ID>*. An example of how

```yaml
---
injection_points:
    - id: '13'
      gadget_name: bp_gadget
      variant: c2
    - id: '14'
      gadget_name: bp_gadget
      variant: c3
```

Listing 4.2: Configuration artifact for the Injection Points. Each Injection Point has a unique numeric ID.

```
#include <iostream>
using namespace std;

int main(){
..

    #IP_14
    for(i=0; i<r; i++){
        #IP_13

        for(j=0; j<c; j++){
            mul[i][j]=0;
            for(k=0; k<c; k++){
                mul[i][j] = (m1[i][k] * m2[k][j]) + mul[i][j];
            }
        }
    }
..
}
```

Listing 4.3: Injection Points embedded in a template configurable system. Each injection point includes its unique ID and a fixed suffix. They are replaced during weaving, with function calls that "invoke" specific gadget variants. Weaving will also add the correct header files to link the shared libraries that constitute gadget variants.

the injection points are specified is shown in Listing 4.2, whereas their actual use in a template system is shown in Listing 4.3. An injection point can appear only once in the code of a template system, and it can map to only one gadget variant. Conceptually, this is consistent with our definition of ISG, but at the same time more practical. This way, modifying an ISG can be done simply by modifying the entries in Listing 4.2 so that an injection point maps to another gadget variant. To ensure this fully matches the ISG definition introduced in Chapter 3, we maintain some additional documentation w.r.t. the specific

in-code locations of each injection point, and encourage users to modify an ISG only by changing the specification mentioned previously, and not by relocating injection points.

On the basis of proper specification and placements of injection points, the Weaver component parses the template systems, captures the injection points, extracts their numeric ID, and finally injects code that invokes the correct gadget variant - at the time of weaving the gadget variants are assumed to have been compiled in the form of shared libraries (cf. Section 4.2). Moreover, each compiled variant will have been assigned values for its knobs as shown in Listing 4.1. After all the above steps are executed, the Weaver performs a second-level derivation through which specific bindings for the features of a template system are enacted, and eventually a variant of the template system is generated and linked with the gadget variants to produce a final executable. This way, the Weaver component handles not only the weaving but also manages the resulting benchmark SPLs. Naturally, any relevant specification for this second-level derivation needs to be provisioned through the Config Registry.

4.2 GADGETS

The gadget SPLs are crucial to our goals, since they serve as the basis for our configurability-inspired approach for constructing synthetic benchmarks. To date, we have implemented two gadgets that target two types of architectural resources: the caches and the Branch Prediction Unit (BPU). Following the approach outlined in Chapter 3, we were able to pinpoint resource-specific metrics[2] of interest and obtain sizeable and consistent impacts w.r.t. those metrics, for each gadget. However, before diving into the specifics of how these gadgets were implemented, we need to describe how gadget derivation works in a bit more detail.

4.2.1 *Derivation*

Each gadget's implementation constitutes two main artifacts, namely the source implementation, and the so-called *binding layer*. The first of the two artifacts comprises the source code which is structured based on the features that we outline in a corresponding feature-model[3].

The second artifact comes in the form of a Makefile - a utility that helps automate compilation tasks. In this artifact we incorporate some logic that enables an easy specification of a feature-selection by the Gadget Manager component we introduced in the previous section, such that a desired gadget variant can be derived, as shown in Figure 4.2. In addition, the binding layer uses unique identifiers for each valid configuration of its features, and consequently for each potential variant in the respective SPL. This makes it possible for the Gadget Manager to keep track of variants, as well as handle user-provided specifications more easily. As such, the provisioning of a variant ID enables an initialization of specific macros in the binding layer. These macros constitute configuration options in terms of the source code artifact, and an initialization of such macros constitutes a feature-selection. To be more precise, we use multiple preprocessor macros (cf. Chapter 2) in the source code of a gadget, with the aim of attaining variability while reusing the same code artifact. This way,

2 Hereafter we refer to these as simply *metrics*, unless we need to distinguish them from other types.
3 The gadgets are implemented in C++.

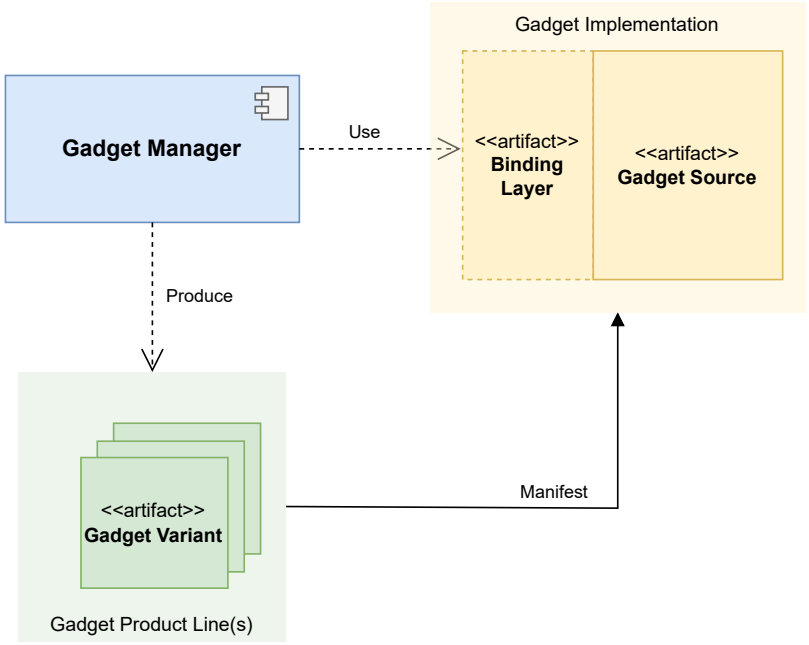

Figure 4.2: Depiction of the Gadget Manager's interaction with the gadget implementation. Notably, the gadget implementation provides a binding layer as an interface to the Gadget Manager, through which important decisions regarding feature-selections and subsequent in-code bindings are handled. The actual implementation of features is contained in the source artifact.

once the Gadget Manager provides a variant ID, the binding layer will invoke the compiler and input the correct values for these macros. Thus, the binding of the selected features will occur during the compilation[4], eventually resulting in the derivation of a desired variant in the form of a shared library. The main reason for performing compilation-time bindings stems from the fact that the we intend to minimize the complexity in the design of the gadgets by not including runtime ideas, though we intend to expand on this aspect in future-work (cf. Chapter 7).

4.2.2 *The Cache Gadget*

Following the approach in Chapter 3, having a gadget that can simulate synthetic behaviors that directly relate to the memory subsystem is beneficial due to the fact that many performance-faults in software systems relate to memory usage patterns [19], and the fact that memory components are widely considered to be a cause for performance bottlenecks [21]. In particular, Drepper [19] outlined several performance-optimization approaches that specifically relate to the way a program makes use of the caches, and how it organizes memory accesses. Based on those and other findings, as well as current features and limitations in modern x86 implementations [20], we were able to build small units of workload that saturate caches on all levels, as well as bypass them in locality-reducing ways. Organizing these findings, we were able to construct the feature-model shown in Figure 4.3.

4 Technically, it will occur during the preprocessing phase that precedes the actual compilation.

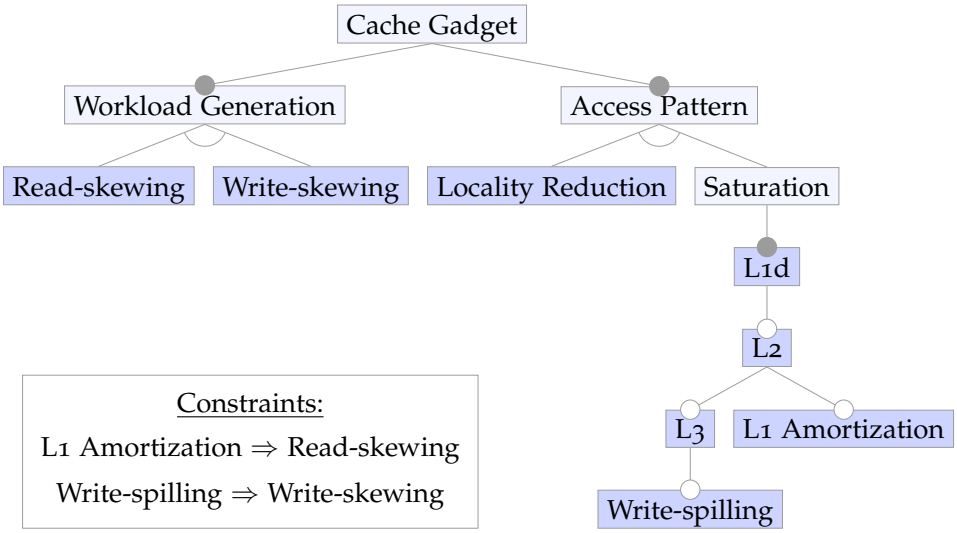

Figure 4.3: Complete feature-model for the Cache Gadget. The indicated constraints can be read as logical implications, e.g. the selection of Write-spilling implies the selection of Write-skewing.

The first step in the design of this gadget was to determine whether we could attain noticeable effects in terms of misses, and more importantly in terms of the miss-rates (provided that there is a substantial amount of accesses performed overall), across all available data caches. The Performance-Monitoring Unit (PMU) of an x86 implementations provides counters that allow us to obtain such metrics for each level of cache. In addition, the available counters distinguish between misses that correspond to read instructions from those that correspond to write instructions. We make use of that distinction in our implementation since that allows us to achieve different miss-rates in the caches for either of the two types. A list of the metrics we use is shown in Table 4.1. While we could include and target many more metrics in our implementation, we opted to focus more on precision and correctness, consequently focusing on metrics for which the underlying counters would be obtained reliably in different systems[5].

The distinction between reads and writes leads us to the first abstract feature, namely the Workload Generation. This is a mandatory feature, since the gadget will necessarily be able to generate workloads. However, we can choose either a read-heavy or a write-heavy workload. Read-heavy workloads focus on reading from memory, whereas the write-heavy workloads perform repeated writes in large memory allocations. Regardless of the specific type of workload in this context, the most important part of the feature model is the manner in which memory accesses are performed. Following a Feature-Oriented Software Development (FOSD) approach, we abstract this idea and make it into an explicit, mandatory, and abstract feature. While this feature is mandatory, we have a choice regarding the specific pattern we want to use in that we can choose either Locality Reduction or Saturation, modeled as child nodes in Figure 4.3.

The Locality Reduction feature relies on an aggressive reduction in temporal and spatial locality, in all levels of caches. Specifically, we rely on some functionalities known as

5 See Chapter 5 for a list of systems in which we run the gadgets.

Metric	Description
L1 Load Miss-Rate	The rate in which misses occur in the first-level data-cache, obtained by the ratio of misses and total cache references, concerning only memory reads.
L2 Load Miss-Rate	The rate in which misses occur in the second-level cache, obtained by the ratio of misses and total cache references, concerning only memory reads.
L3 Load Miss-Rate	The rate in which misses occur in the third-level cache, obtained by the ratio of misses and total cache references, concerning only memory reads.
L3 Store Miss-Rate	The rate in which misses occur in the third-level data-cache, obtained by the ratio of misses and total cache references, concerning only memory writes.
Load Page-Walks	The number of initiated page walks that happen as a result of misses in virtual-address translation, across all TLB levels. The misses are counted only for memory reads.
Store Page-Walks	The number of initiated page walks that happen as a result of misses in virtual-address translation, across all TLB levels. The misses are counted only for memory writes.
CPI	The number of CPU cycles spent in executing an instruction, on average. Usually obtained by dividing the number of CPU-cycles spent executing a process, including stall cycles, and the number of retired (completed) instructions for that process.

Table 4.1: Metrics used to quantify the behavior of the Cache Gadget.

streaming writes and *cache-flushing* [19, 20], offered by x86 implementations in the form of native instructions of the Instruction Set (IS). The first functionality enables memory writes that do not go through any of the caches, presumably because the affected items in memory will not be used soon after, whereas the second one enables us to flush entire cache-lines from the caches in ways that reduce overall locality. Conveniently, we make deliberate use of these functionalities, forcing misses across the cache hierarchy as well as significant latencies. For instance, we can perform a streaming write in some memory location, and then try to read that data immediately after. Because we bypass the cache during the write, the data that was written to memory will not be present in the caches, and so the subsequent read will suffer from cache misses. Similarly, we can repeatedly flush cached data, and immediately after attempt to read that data. Such a read will cause misses in the caches and will have to go to the main memory.

The Saturation feature also makes use of locality ideas, though it achieves its purpose without using any intrinsic instructions of x86 systems. Its aim is to generate workloads that exceed the cache sizes progressively, which will enable two things: 1) the caches will suffer from misses, and 2) prefetching at the hardware level will be rendered sub-optimal.

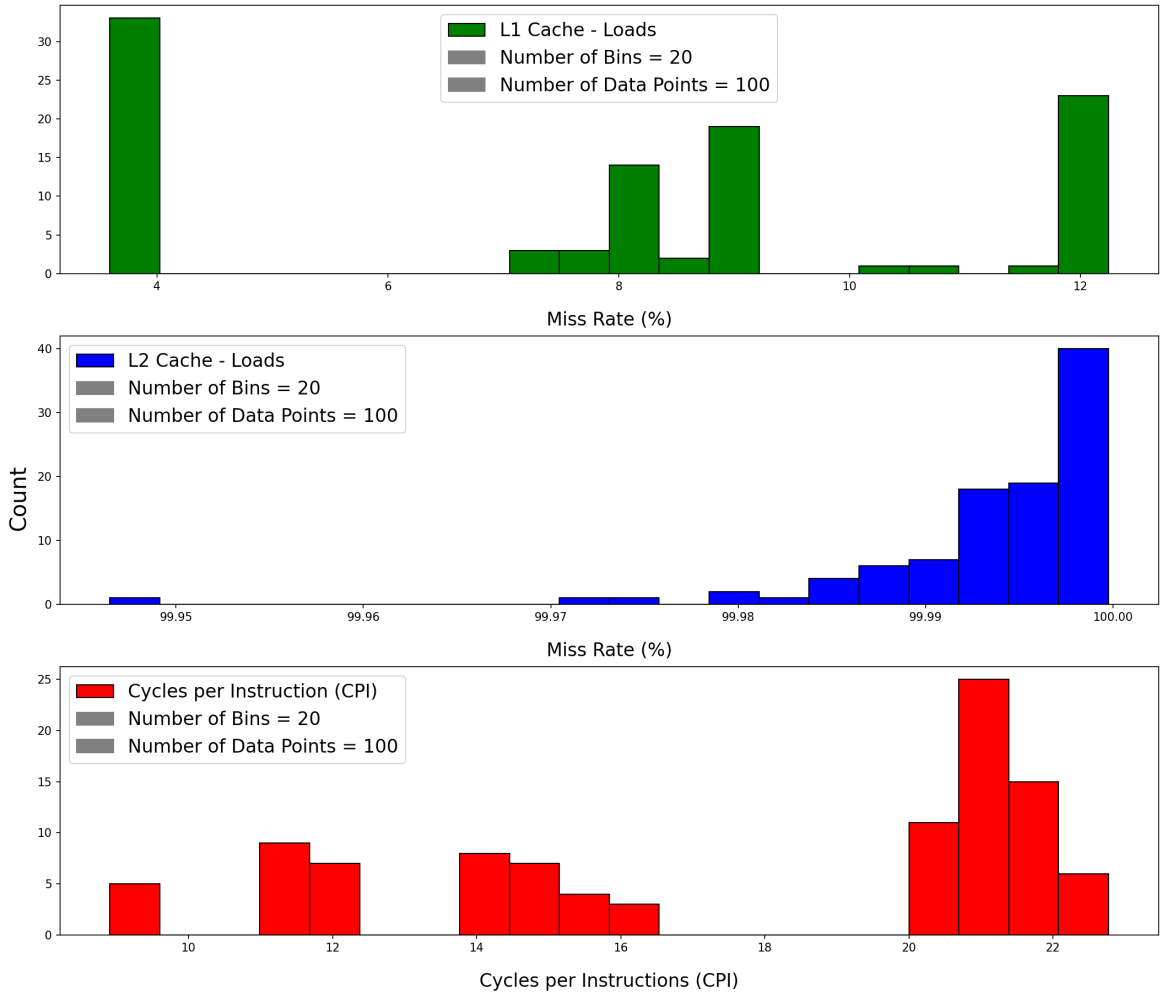

Figure 4.4: A histogram of frequency distributions for samples of values of L1 Load Miss-Rate, L2 Load Miss-Rate and CPI. The number of bins used was set to 20 as it better captured the overall pattern of the distribution. The measurements were performed in a reference architecture, which we introduce during evaluation Chapter 5.

The first situation occurs due to multiple factors that also relate to the use of Translation Lookaside Buffers (TLBs). To begin with, the misses occur because we make deliberate use of little to no locality, and repeatedly access data that are not cached. At the same time, the caches are saturated as they attempt to maintain as many data items as possible to increase the likelihood of hits, hence the name of the feature. In addition, we deliberately access data spanning different pages of memory which naturally forces many misses in the TLBs and increases latencies. To some extent, we also attempt to simulate cases of the aliasing effects we mentioned in Chapter 2, considering how virtually all modern x86 implementations rely on set-associative caching and the fact that some Intel systems map memory addresses to sets in a somewhat fixed manner, allowing us to impact the same sets repeatedly. The second situation then occurs due to an inherent limitation in how virtual address translation is handled. Specifically, the limitation consists in a fixed boundary on the (virtual) addresses that the prefetcher component of a Central Processing Unit (CPU) can use, which is usually

Knob	Values	Description
WSS_SCALER	$\{1, 2, 4, 8, 16\}$	WSS denotes the size of the memory region touched on one iteration of the internal workload loop of the gadget. WSS_SCALER simply multiplies the value of WSS using one of the values shown on the left. The effect of this knob varies with the variants, but it usually tends to reduce the rate of TLB misses, which improves CPI.
OPS_SCALER	$\{10^n \mid n \in \{1, 2, 3, 4\}\}$	OPS denotes the overall number of iterations performed by the internal workload loop of the gadget. The default number can be scaled up using this knob. We opt to use only multiples of ten so as to have noticeable effects when tuning this knob.
ALLOC_SCALER	$\{1, 2, 4\}$	The gadget typically allocates memory sections before applying a workload. This knob helps change the allocation size, which can contribute to a slightly different rate of misses in the caches, in relation to the default allocation size. This is mostly useful when the overall allocation size needs to exceed that of a given cache by more than a small percentage.

Table 4.2: Knobs used in the Cache Gadget for fine-tuning of impacts.

restricted to a single page (4096 bytes) in many Intel x86 systems [20]. In simple terms, this means that prefetching cannot be performed for data items that reside more than a page-size away from currently cached data, in terms of the respective memory addresses.

In either scenario, the effects can be felt progressively through all levels of caches. For instance, we can tune certain parameters so as to impact only the Level-1 data cache. Tuning these parameters further can result in both Level-1 data and Level-2 caches incurring high miss-rates. To obtain a case in which we incur high miss-rates at Level-2, but not at Level-1, we introduce the L1-Amortization feature which improves the locality behavior at the Level-1 data cache, though this is only possible for read-heavy workloads in our current implementation. Using a similar approach, when we opt to impact Level-3 caches in a write-heavy workload we can significantly decrease the number of writes made in the same memory page while still performing the same amount of writes overall, thus incurring even higher TLB miss-rates and much higher CPI as a result of an increased numbers of page-walks (cf. Chapter 2). This last idea is captured by the Write-spilling feature.

To depict all these aspects, we use the Reference Impact-Mapping (IM_{ref}), introduced in Chapter 3. A partial example for a gadget variant is shown in Figure 4.4, using only some of the relevant metrics. We specifically plot a histogram of the values of the targeted metrics for a specific variant, and knob assignment.

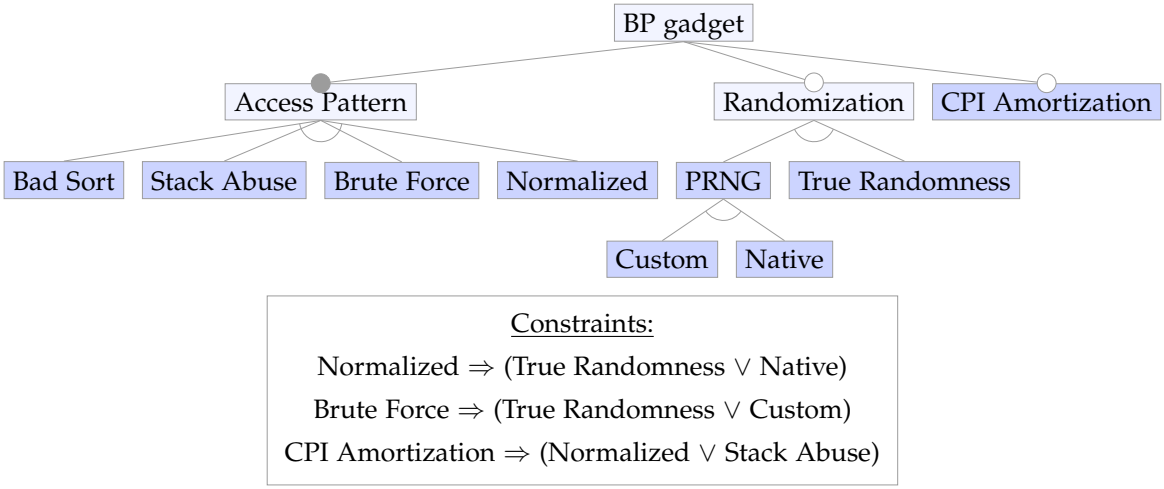

Figure 4.5: Complete feature-model for the Branch Prediction Gadget. Constraints should be read as logical implications.

Lastly, we present the knobs of Cache Gadget together with their value domains in Table 4.2. We also describe their intended use; overall, the knobs of the Cache Gadget serve to scale the applied workload. These knobs are available for each variant, and they can usually alter the behavior of each such variant. In any case, our implementation includes safeguards that try to prevent knobs from altering the behavior of a variant to the extent that would make the same variant attain a behavior that is too similar to that of another variant. In other words, for each variant we try to maintain some distinct profile, and knobs should not affect this.

4.2.3 *The Branch Prediction Gadget*

While the Cache Gadget provides us with a way of synthesizing scenarios of bad cache locality or aggressive memory usage, we can also target computational components of an architecture. Most x86 implementations provide counters regarding the overall performance of a program w.r.t. the CPU, especially in terms of the pipeline bubbles and the ensuing stalls. On this basis, we developed a gadget that deliberately introduces stalls, by focusing on the workings of the BPU. In particular, we attempted to synthesize workloads that attain a high level of branch-mispredictions, given their importance in the performance of software in modern processors [33, 34]. As a result, we managed to design the feature-model depicted in Figure 4.5.

The feature-model has two main parts, denoted by the abstract features Access Pattern and Randomization. The Access Pattern implies the manner in which a workload accesses caches and memory. However, in this context the feature refers more to the way instructions are accessed from the Level-1 instruction cache. Furthermore, we have several, mutually exclusive choices in terms of access patterns: Bad Sort, Stack Abuse, Brute Force and Normalized. The Bad Sort feature corresponds to a badly implemented sorting algorithm,

Metric	Description
Branch Misprediction-Rate	The ratio of branch mispredictions and overall branching instructions. Branching instructions can be conditional (if-else types) or simple jumps (goto types).
Conditional Branch Misprediction-Rate	The ratio of branch mispredictions and overall branching instructions, considered only for conditional branching instructions.
Front-end Boundedness	Shows the ratio of stalls that occur due to front-end bubbles, over total stalls.
Bad Speculation	Shows the percentage of unsuccessful speculative executions after a branch prediction. The higher the percentage, the more effort is wasted by the CPU due to a misprediction (on average).
CPI	The number of CPU cycles spent in executing an instruction, on average. Usually obtained by dividing the number of CPU-cycles spent executing a process, including stall cycles, and the number of retired (completed) instructions for that process.

Table 4.3: Metrics used to quantify the behavior of the Branch Prediction Gadget.

which performs multiple redundant passes over elements in an array without actually sorting correctly, resulting in unnecessary branch mispredictions. The Stack Abuse feature corresponds to a tail-recursive implementation that presents a specific challenge to the BPU, in that the return path of a recursive function call contains branches that can be easily mispredicted, mostly because the return path is too long and many return addresses are not kept in the call-stack, thus becoming difficult to predict.

The other two access patterns are a bit more involved, as they include the use of deliberate randomization strategies, which is why we have an abstract Randomization feature in the feature-model. This feature is broken down into two possible implementations of the Random Number Generators (RNGs) that can be used. The first implementation is a Pseudo-Random Number Generator (PRNG), whereas the second one makes use of a specific x86 instruction that obtains random numbers from an RNG that is seeded with hardware-level entropy. The PRNG implementation usually involves the use of the default standard-library implementation. However, we provide a custom Linear (Congruential) Random Number Generator (LRNG) implementation, since the standard implementation appeared a bit inconsistent during our initial tests. With the help of randomization, we were able to implement access patterns that incur high branch-misprediction rates. This was made possible due to the fact that we execute the same branch instructions repeatedly, and in each execution the branch decision is random, thus making the prediction difficult for the BPU. Indeed, the goal is for the BPU to not be able to figure out a recurring sequence of decisions w.r.t a given branch instruction, and for it to resort to simple guesses. When such guesses are wrong, bubbles can form in the pipeline, causing significant latencies. Both the Brute Force and Normalized features incorporate this idea, but the former also incorporates deliberate loop-unrolling so as to reduce the number of non-conditional branches (e.g. branches that

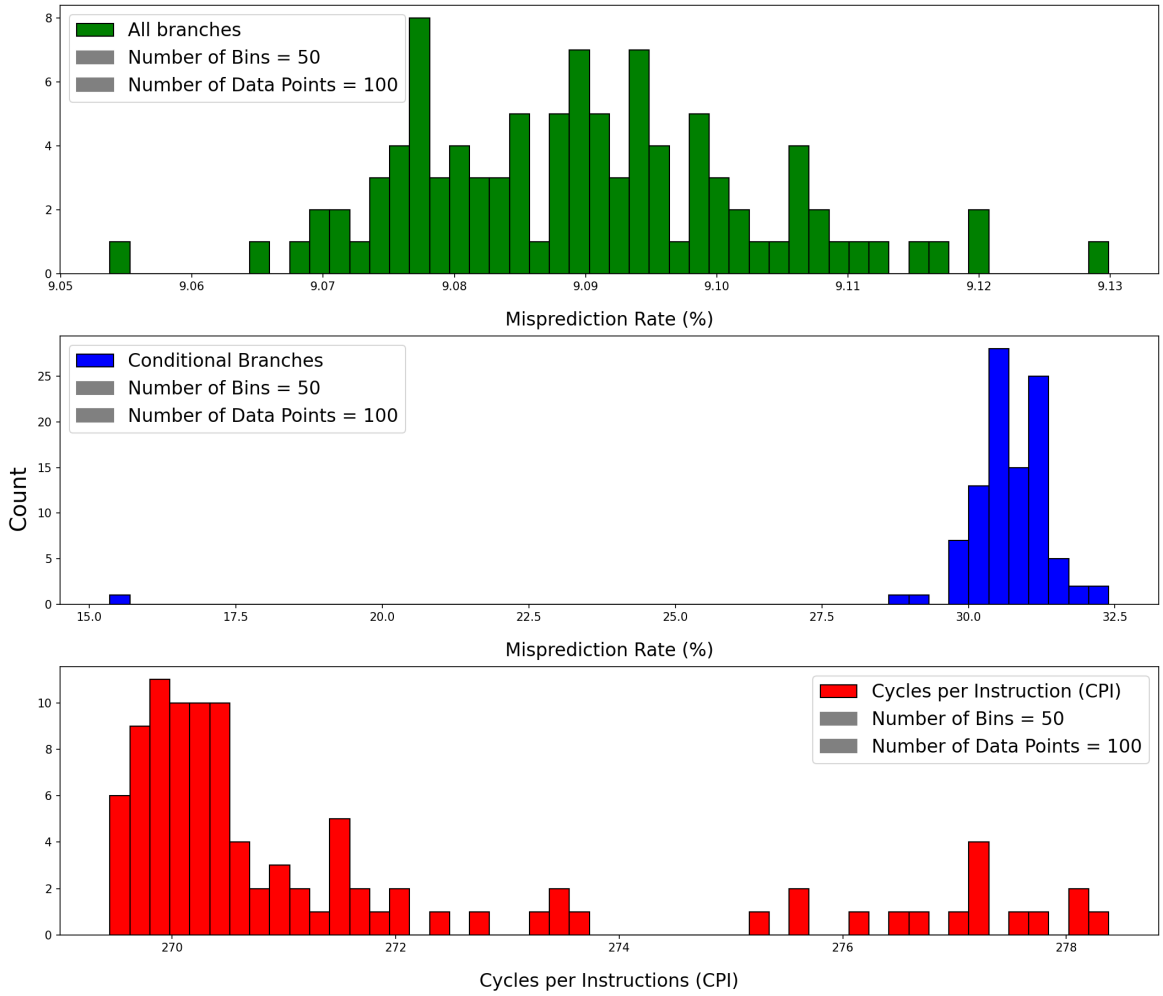

Figure 4.6: A histogram of frequency distributions for samples of values of Branch Misprediction Rate, Conditional Branch Misprediction Rate and CPI is shown. The number of bins used was set to the large value of 50, since lower values smoothen the overall shape significantly. The measurements were performed in a reference architecture, which we introduce during the evaluation in Chapter 5.

are not if-else conditions), thus providing us with a distinct behavior. Non-conditional branches are always "taken", and their predictability relates to the *target prediction* idea discussed in Chapter 2. Besides the randomization ideas, these two patterns are effective due to an imminent exhaustion of the capacity of Branch Target Buffer (BTB). We ensure this saturation by incorporating a large number of branches, and ensuring that the same branch instruction is not re-executed too soon (i.e. we reduce temporal locality w.r.t. the BTB). Having a saturated BTB implies that additional latencies can be accrued because the prediction cannot make use of any Pattern History Table (PHT) entries, which can also lead to bubbles.

The last feature in the feature-model is CPI Amortization. This feature provides us with added variety in terms of the behaviors we obtain, though it simply helps reduce the overall latencies that are accrued by mispredictions. Note that the feature-model also incorporates several constraints for some features. Notably, the use of Normalized or Brute Force patterns

Knob	Values	Description
OPS_SCALER	$\{10^n \mid n \in \{1,2,3,4\}\}$	OPS denotes the overall number of iterations performed by the internal workload loop of the gadget. The default number can be scaled up, using this knob. We opt to use only multiple of tens so as to have noticeable effects when tuning this knob.
ALLOC_SCALER	$\{1,2,4\}$	The gadget typically initializes simple arrays with specific sizes. This knob helps change the array size, which simply increases the overall number of instructions, potentially providing better resolution (less noise) in terms of the metrics that we need to measure.
LOPSIDE	$\{0,1,2\}$	In cases when Randomization is used, the amount of randomness the conditional branching instructions have to endure can be varied using this knob. Value 0 corresponds to full randomization, in which a given conditional branch has an approximate 50% chance of being predicted correctly. Values 1 and 2 correspond to the approximate cases of 25% and 12.5%, respectively.

Table 4.4: Knobs used in the Branch Prediction Gadget for fine-tuning of impacts.

implies the activation of the Randomization feature, by design. In addition, the Brute Force feature can only use the custom LRNG implementation because the customizability of the LRNG can be beneficial in case the overall workload results in very high latencies.

While the access patterns described earlier largely follow the same ideas, we have managed to ensure that different variants of the gadget can lead to distinctly varied behaviors. Based on our approach, we make use of the metrics shown in Table 4.3 to verify this. The overall rationale in choosing these metrics is that we want to see not only high misprediction rates, but also distinguish between conditional-branches and non-conditional branches. Furthermore, characterizing the types of bubbles we observe is also compelling, which is why we try to distinguish between bubbles that happen in relation to the front-end of the CPU, as opposed to the back-end (cf. Chapter 2). In Figure 4.6 we provide a partial example of the IM_{ref} for a specific variant.

Lastly, the Branch Prediction Gadget also incorporates knobs, as shown in Table 4.4. The use of knobs in this case is mostly analogous to that of the Cache Gadget, though slight differences exist. One difference is that some knobs are more variant-specific; for instance, the Lopside knob is only useful for variants of the gadget that include the Normalized feature, whereas for other variants it simply reverts to a default value without making any difference. To give a better idea of how knobs can help, we provide an example in Figure 4.7, where we can see that the knob can help us manipulate the rate of mispredictions in a meaningful way. Specifically, we make use of a variant of the Branch Prediction Gadget

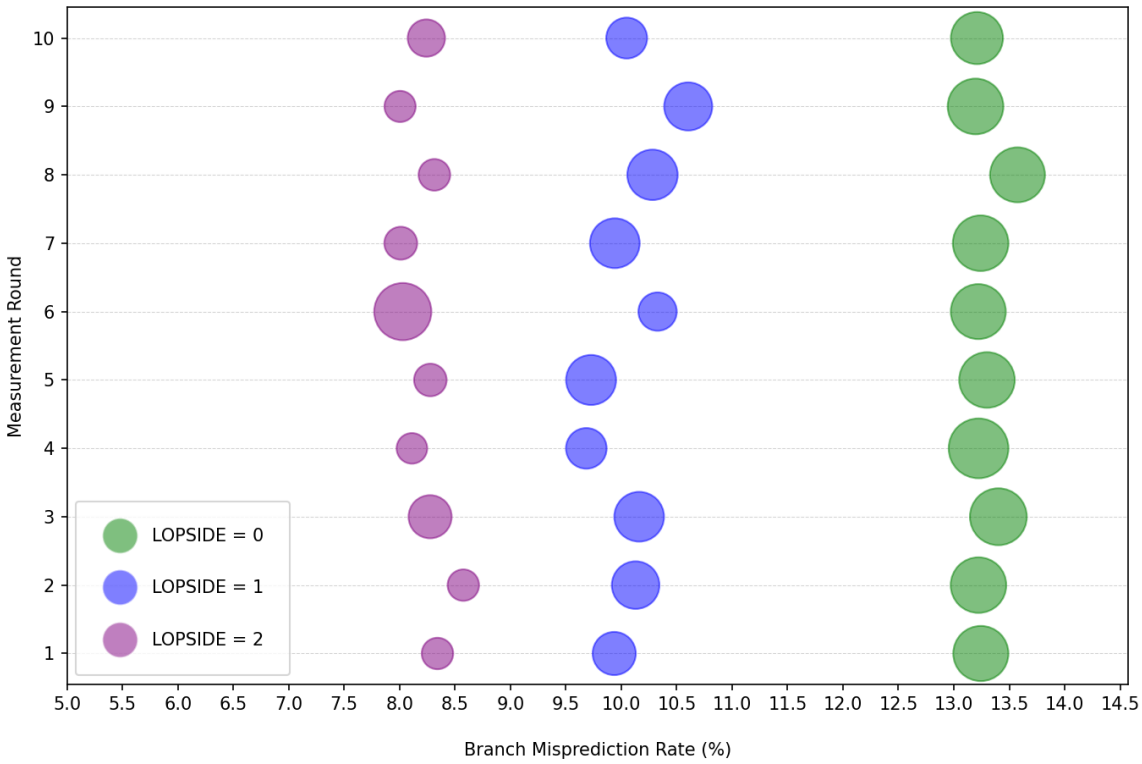

Figure 4.7: A bubble plot indicating the effect of the Lopside knob. We perform 10 rounds of measurements. In each round, we take a measurement for each of the three knob values. The measurement rounds are shown in the y-axis. The size of each bubble (data point) indicates the value of the CPI obtained in the same round, relative to the other bubbles. The observed behavior is consistent with the description in Table 4.4.

and try out different values for the Lopside knob. After having taken a measurement for each value of that knob, we repeat the overall process. In total, we make 10 repetitions. The reason for doing this in rounds, rather than taking 10 measurements at a time for each knob value, is that across rounds, the system-wide CPU utilization is varied so that we can see the knobs at work in different conditions[6].

4.3 CONFIGURABLE SYSTEM TEMPLATES

To facilitate the actual synthesis for benchmarks, we rely on configurable systems that serve as templates for the injection of gadget variants. The code artifacts of these templates are augmented with injection points by design. These injection points can be moved to different locations, but we insist on specifying them in a careful manner and considering them as an integral part of the design of a template system. This way, if the ISG has to be modified for a template system, we should simply change the way injection points map to gadget variants. Furthermore, each template system has its own *binding layer* through which we can obtain different variants thereof; this is something that the Weaving component handles, similarly to how the Gadget Manager handles the gadget SPLs. Moreover, while we can

6 We isolated the gadget variant only in the first round.

obtain variants of a template system, any such template will not be functional unless the processes described in Figure 3.3 are executed. Specifically, the gadget variants need to be derived first, and then weaving has to be performed on the basis of an (ISG) that clearly maps the injection points to gadget variants.

An important aspect regarding the template systems we have implemented is that they can help us test and evaluate the overall approach, but can also be of direct use to researchers. However, the templates cannot be assumed to be representative of existing systems, which means that we do not claim that our approach can induce faults in actual systems, though this is something we intend to pursue in future work (cf. Chapter 7). Our templates typically have some workload that allows us to obtain pre-injection measurements which can be used as baselines. In addition, the templates typically indicate different yet low levels of CPU- or Memory-boundedness for different variants, by design[7]. For instance, we have built a template called Scrambler that performs mathematical operations such as Discrete Fourier Transforms, while operating on scrambled inputs which it stores in and reads from memory. Focusing on mathematical operations means that the template's inherent workload could be CPU-bound if intended, in the sense that the latency accrued for that workload could be made to depend on the frequency of the CPU, rather than memory-related latencies. Similarly, we have implemented a template called Memo that could be made memory-bound by emphasizing memory access latencies. However, we make sure that by default, neither of these templates comprises variants that indicate such characteristics.

4.4 LIMITATIONS AND DETAILS

Our implementation realizes the approach we outlined in Chapter 3. However, it has several limitations and is largely a work in progress. Some important limitations are as follows. Firstly, we have inadvertently tailored the gadgets to x86 implementations, and in particular Intel ones. The primary reason for this is that x86 implementations provide an extensive instruction set which we take advantage of for things such as cache bypassing. In addition, we have relied mostly on Intel-specific manuals and literature to inform the design of our gadgets, and are cognizant of the fact that our gadgets might not work well in x86 implementations such as AMD. Secondly, in our implementations we deliberately disable any compiler optimizations, as they interfere with our implementation in very significant ways. For instance, we found that the use of any form of vectorization by the compiler removes much of the impact that the Cache Gadget is supposed to attain.

On the level of the framework, the main limitation relates to the fact that both gadgets and template systems are exclusively implemented in C++, with no support for other languages. Furthermore, extending the framework with more gadgets is not trivial because of the need for careful consideration of the binding layer introduced earlier, as well as namespacing issues that need to be accounted for during the weaving process[8]. Furthermore, the overall workload mixing that can be attained by injecting gadget variants has only been considered in terms of a sequential execution flow, and it is entirely static. We intend to improve on this aspect in future work (cf. Chapter 7).

7 Please refer to Appendix A for the feature-models of the two template systems.
8 These issues are documented in the code repository.

Lastly, an important limitation exists in terms of the semantics of gadgets. Considering that what is injected into a template is an already derived gadget variant, that inherently means that there is no way of dynamically tuning the gadget at runtime. Nevertheless, this can make each injected variant an atomic unit of workload, which helps in the overall synthesis of behaviors in the context of the template systems.

5 EVALUATION

In this chapter, we evaluate the approach presented in Chapter 3, on the basis of our implementation of the core elements, namely the gadgets and the weaving framework. Specifically, we outline two research questions that relate to the core premises of our approach: ensuring stability in the behaviors of gadgets and providing a practical way of creating synthetic benchmarks that can be helpful to researchers.

5.1 RESEARCH QUESTIONS

To evaluate our work, we answer the following research questions:

- **RQ1:** Do gadgets provide stable behaviors and thus enable us to precisely and easily define their Ground-Truths?
  - **RQ1.1 (Consistency):** How consistent is the behavior of a gadget, when used in isolation?
  - **RQ1.2 (Reproducibility):** Do the gadgets indicate the same behavior across different environments?
- **RQ2:** Does our approach enable us to produce useful synthetic benchmarks on the basis of gadgets?
  - **RQ2.1 (Inter-Gadget Influences):** Do different gadget variants, when executed together, influence the impacts of one another in ways that alter their individual impacts?
  - **RQ2.2 (Weaving):** Is our weaving approach effective in altering the existing performance profile of a configurable system template?

5.1.1 *Research Question 1*

Our primary concern is to have the gadgets act as reliable building blocks for synthetic benchmarks. For that reason, we outline **RQ1** through which we evaluate whether the gadgets can have a known and stable behavioral profile out-of-the-box. In the same vein, we want to see how feasible it is to define a precise Ground Truth (GT) for each gadget, which is essential to our approach for reasons of stability and transparency, and to see whether we can do so without manually obtaining Impact-Mappings (IMs) for every possible environment (cf. Chapter 3).

To that end, our initial concern is with regard to the behavioral profile of each gadget variant when it executes in isolation, i.e. unaffected by other processes (**RQ1.1**). In Chapter 3 we presented the notions of an IM and Reference Impact-Mapping (IM_{ref}), which help us discern whether the gadgets have consistent impacts with regard to specific resources,

indicated through metrics that were specified accordingly. We presented the relevant metrics for each of the two gadgets we have built in Chapter 4. Thus, we now focus in obtaining a more descriptive view of the IM_{ref} for each of our gadgets. As introduced in Chapter 3, the IM_{ref} comprises a series of aggregate values for each metric of interest, obtained for each variant of a given gadget and for a specific knob assignment, in a reference environment that we choose. Having a high level of consistency in these values can directly help in achieving overall stability, and consequently in being able to pinpoint the GT of each gadget.

Informed by the results of **RQ1.1**, we then try to evaluate whether the gadgets have behaviors that can be reproduced in other x86 microarchitectures (**RQ1.2**). Reproducibility would indicate that the behaviors of gadgets can generalize to different environments, which implies that a gadget's GT could possibly be defined without doing measurements in every environment. However, an important issue arises due to the resource-specific nature of the gadgets, whereby minor differences between microarchitectures can potentially render their implementation impotent in some environments. Naturally, we rely on the IM_{ref} from **RQ1.1** to get some initial expectations, though what we evaluate specifically is whether the IMs we get in different environments resemble one another.

5.1.2 *Research Question 2*

While we can evaluate the behaviors of gadgets in isolation, our ultimate goal is to produce synthetic benchmarks. To that end, we outline **RQ2** with the aim to evaluate the runtime semantics of our gadgets and their usefulness in the weaving process that we outlined in Chapter 3. Firstly, we are interested in knowing whether unexpected behaviors arise when multiple gadget variants are combined in the same execution, i.e. combined in a single running process (**RQ2.1**). The motivation for this stems from the fact that we want our (presumably) stable gadget behaviors to persists in the context of the larger benchmarks we build, so that the benchmarks themselves are stable (cf. Chapter 3). To that end, a cause for concern regarding this case arises from the fact that we have targeted subsystems related to caching and branch-prediction. The inherent intricacies in how these subsystems function pose challenges to the usefulness of our implementations, especially considering that we want to use gadgets as building blocks for the derivation of synthetic benchmarks and thus have no unintended inter-gadget influences.

Notably, we are wary of the hardware prefetching strategies that could be employed by x86 microarchitectures for the caches, as well as the use of global history tables in modern Branch Prediction Units (BPUs). Hardware prefetchers can employ a variety of algorithms, many of which are proprietary and hence undisclosed [20]; regardless, advanced techniques are likely to be used [26], which can be cause for concern in our context. In addition, the use of global histories by the BPU means that correlations between different branching instructions in an executing process are exploited for more accurate predictions [34, 45]. On this basis, we need to evaluate whether any gadget variants experience any reduction in their impacts when combined with other variants in the same execution, as well as to see whether the execution order of multiple gadget variants can be a confounding factor with regard to this issue.

As the last part in our evaluation, we aim to put everything in context and thus try to evaluate the usefulness of our approach through **RQ2.2**. Informed by the results of **RQ1**

and **RQ2.1**, we try to make use of the gadgets to build synthetic benchmarks with distinct behavioral profiles and to discuss what limitations exist. In doing so, we aim to indicate how the notion of an Injective System-Gadget Mapping (ISG), presented in Chapter 3, is useful in obtaining benchmarks with varied behaviors.

5.2 OPERATIONALIZATION

To address the questions outlined above we need to perform measurements and experiments, using different subject systems as test environments. With regard to measurements, we described the overall idea of resource-specific metrics in Chapter 3, and the measurements techniques of counting and profiling in Chapter 2. Relying on the counting of performance events, we are able to measure the impact of each gadget variant and thus obtain series of values for each resource-specific metric that the variant targets, as specified by the $\triangleright$-relation (cf. Chapter 3). This is made possible by counting Performance-Monitoring Unit (PMU) events using Model-Specific Registers (MSRs), and then aggregating the event counts to obtain values for our specified metrics. Overall, we focus on 16 variants of our gadgets in total[1], 7 of which correspond to the Branch Prediction Gadget, and the rest to the Cache Gadget. These variants are listed in Table 5.1.

Importantly, we make sure to account for any potential issues with the precision of the counting of events by isolating the gadgets during execution. We achieve that by setting a high affinity to a random Central Processing Unit (CPU) core so that the counting is not affected by inter-core process migrations[2], as well as by relying on a controlled cluster environment that makes use of *cgroups* and guarantees that any other process running in the system does not affect the gadget process(es)[3]. Overall, we make use of five different environments, four of which are Intel implementations that correspond to generations 4-7 of the Intel processors. For completion, we also make measurements in an AMD system. All the concerned environments are described in Table 5.2, where we also highlight our *reference* environment.

Remark (Relaxed Impact Mappings)
In Chapter 3, we discussed in detail how an IM relates variants of the gadgets to metrics through the *targeting* relation, while accounting for different knob assignments. In order to simplify our evaluation, for every measurement or experiment we use maximal assignments of values for the knobs, such that we obtain the maximal possible impacts for each gadget variant. The rationale for doing this is that knobs provide a useful yet not critical utility, since they do not alter the behaviors of gadget variants that much by design. That means that for our research questions, it is enough to maintain the same knob assignments throughout measurements and experiments as if they were a fixed part of the functional design of each gadget variant. Indeed, we make sure to use the same assignment of values to the knobs for any gadget variant across different measurements and environments. In any case, we intend to evaluate their implications in future work (cf. Chapter 7).

1 We have omitted select variants due to correctness issues we encountered during our work.
2 There have been certain issues in the past relating to imprecise counter values, in the context of hyperthreading. Patches exist but are not integrated in all Linux kernels. An example is given in https://lore.kernel.org/lkml/1416251225-17721-7-git-send-email-eranian@google.com/.
3 *cgroups* are a Unix-system utility that enables performance isolation based on user permissions.

Gadget	Vnt.	Selected Features
BP Gadget	v_1	BP Gadget, Access Pattern, Normalized, Randomization, PRNG, Native
	v_2	BP Gadget, Access Pattern, Stack Abuse
	v_3	BP Gadget, Access Pattern, Normalized, CPI Amortization, Randomization, PRNG, Native
	v_4	BP Gadget, Access Pattern, Stack Abuse, CPI Amortization
	v_5	BP Gadget, Access Pattern, Brute Force, Randomization, PRNG, Native
	v_6	BP Gadget, Access Pattern, Normalized, Randomization, True Randomness
	v_7	BP Gadget, Access Pattern, Brute Force, Randomization, PRNG, Custom
Cache Gadget	v_1	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Saturation, L1d
	v_2	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Locality Reduction
	v_3	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Saturation, L1d, L2
	v_4	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Saturation, L1d, L2, L1 Amortization
	v_5	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Saturation, L1d, L2, L3
	v_6	Cache Gadget, Workload Generation, Read-Skewing, Access Pattern, Saturation, L1d, L2, L3, L1 Amortization
	v_7	Cache Gadget, Workload Generation, Write-Skewing, Access Pattern, Saturation, L1d, L2, L3
	v_8	Cache Gadget, Workload Generation, Write-Skewing, Access Pattern, Locality Reduction
	v_9	Cache Gadget, Workload Generation, Write-Skewing, Access Pattern, Saturation, L1d, L2, L3, Write Spilling

Table 5.1: Gadget variants and the corresponding feature-selections. The feature-models for each gadget were introduced in Chapter 4.

Therefore, for any gadget $g \in G$, we will only use the relaxed forms of IM and GT, introduced in Chapter 3:

$$\mathcal{IM}_g^{\mathcal{B}} : (\mathcal{V}_g \triangleright \mathcal{M}_g) \to \mathcal{R}_0^{+^n} \tag{5.1}$$

$$\mathcal{GT}_g^{\mathcal{B}} : \mathcal{E} \times (\mathcal{V}_g \triangleright \mathcal{M}_g) \to \mathcal{R}_0^{+^n} \tag{5.2}$$

where $\mathcal{B} \subseteq \mathcal{A}_g$ is a set of *maximal assignments*, in the sense that for each variant $v \in \mathcal{V}_g$, it includes an assignment for which we can obtain the highest values possible w.r.t. the metrics

#	Microarchitecture	CPU	Caches	Main Memory
1	Intel Skylake - 6th Generation	2.60 GhZ, 6-core, 64-bit	L1d: 32 KiB, per-core. L2: 256 KiB, per-core. L3: 12 MiB, shared.	64 GiB, DDR4
2	Intel KabyLake - 7th Generation	3.20 GhZ, 4-core, 64-bit	L1d: 32 KiB, per-core. L2: 256 KiB, per-core. L3: 6 MiB, shared.	16 GiB, DDR4
3	Intel Broadwell - 5th Generation	3.10 GhZ, 10-core, 64-bit	L1d: 64 KiB, per-core. L2: 512 KiB, per-core. L3: 50 MiB, shared.	256 GiB, DDR3
4	Intel Haswell - 4th Generation	3.30 GhZ, 4-core, 64-bit	L1d: 32 KiB, per-core. L2: 128 KiB, per-core. L3: 6 MiB, shared.	16 GiB, DDR4
5	AMD Zen3	4.10 GhZ, 8-core, 64-bit	L1d: 32 KiB, per-core. L2: 512 KiB, per-core. L3: 64 MiB, shared.	256 GiB, DDR5

Table 5.2: Subject systems used as test environments for our measurements and experiments. The green-shaded row denotes the reference environment. Note that the L1d and L2 caches sizes are shown as individual (per-core) sizes. The overall L1d and L2 capacities can be found by multiplying the given values with the number of cores.

that v targets. From now on, we imply the use of the relaxed form whenever mentioning IM, IM_{ref} and GT. The knob assignments are thus implied in all cases.

5.2.1 *Research Question 1*

In order to tackle **RQ1**, we perform different experiments or measurements for each of its sub-questions. Regarding **RQ1.1**, we perform measurements for each gadget variant; specifically, we obtain independent samples of values for the metrics that are targeted by the variant. For completion, we also obtain values for metrics that are not targeted by a variant, as long as these metrics correspond to the gadget overall. However, targeted metrics are in our main focus. In any case, we ensure the independence of these samples by introducing short wait-times between executions. In each execution, we are able to obtain one aggregated value for each targeted metric. That means that across multiple executions, we are able to get entire samples of values for each such metric, conforming to the notion of IM.

Naturally, the metric values we obtain are aggregated from counts of PMU events that we measure (count) for each specific variant, during its execution. In order to count a sufficient number of PMU events, it is important that we allow the gadget variants to run for an adequate amount of time - otherwise, since there is a limited number of MSRs that can be used for counting, certain events might be under-counted. The maximal knob assignments are particularly helpful in this regard. To verify the correctness of our measurements we also perform profiling for each variant, making use of Precise Event-Based Sampling (PEBS), a feature offered by most x86 implementations through which event counts are measured

with high precision and combined with trace data. After obtaining accurate event counts, we aggregate the results and obtain values for our target metrics based on simple calculations that use the event counts. In the context of **RQ1.1**, we rely on our *reference* environment for these measurements and thus obtain an IM_{ref} for each of our two gadgets.

Practically, any IM yields a superset of frequency distributions - for each gadget variant, we have multiple frequency distributions (one for each targeted metric). Since the IM_{ref} is intended as an important description of a gadget's design, we make it more tangible by reporting both the means and the Coefficient of Variation (CoV) of each frequency distribution. Moreover, considering that the frequency distributions can present varying levels of skew or kurtosis, we derive bootstrapped confidence-intervals on the basis of resampling [46]. The bootstrapping incorporates bias-correction analysis, which accounts for high levels of skew in the distribution and improves the accuracy of the confidence-intervals. In addition, we perform goodness-of-fitness test regarding the normality of the frequency distributions using the Shapiro-Wilk method [47].

To address **RQ1.2**, we obtain an IM in each of the environments listed in Table 5.2. An important limitation we faced is the fact that the underlying PMU counters we use in Intel systems have significantly different semantics than those in AMD systems (cf. Section 5.5). In spite of this, we are still able to perform exact comparisons among the Intel environments. After having obtained the IM_{ref} in **RQ1.1**, we obtain an IM in each of the other three Intel environments, for each gadget. We then attempt to evaluate whether statistically significant differences exist with regard to the (shapes of) frequency distributions in the IMs obtained across all four environments, for each gadget. We rely on the Kruskal-Wallis test [48]; using a non-parametric test is important due to the normality results we obtained beforehands, which we also obtain for the non-reference, Intel environments.

5.2.2 *Research Question 2*

The second question we intend to address relates directly to the practical use of our benchmark-producing framework (**RQ2**). To that end, we firstly design an experiment that helps us address **RQ2.1**. Specifically, we make use of a configurable system template that has no intrinsic workload of its own, called the Empty Template. We introduce two injection-points in this system in order to combine pairs of gadget variants in the same execution. Notably, we only consider pairs of variants that belong to the same gadget due to the fact that our rationale for this sub-question stems from ideas that apply either to the caches or the BPU, as described previously. The Experiment component introduced in Figure 4.1 has been designed to produce appropriate ISGs for this case. Overall, the goal of the experiment is to determine whether the impacts obtained by a given pair deviates from predefined expectations that are calculated from the impacts of each variant in isolation (cf. **RQ1.1**).

For each such pair, we perform measurements similarly to the ones described for **RQ1.1**. However, an important detail for this experiment is that we consider the *intersection* of the metrics that are targeted by the injected gadget variants. The reason is that we want to define clear expectations for the values these metrics can take, which would not be the case if one of the variants does not explicitly target a given metric while the other does. In fact, if a metric is not targeted by a variant, that variant can obtain highly-varied values w.r.t. that

metric (cf. Chapter 3). Moreover, we only consider rate (percentage) metrics. With these in mind, for any two variants v_1 and v_2 of the same gadget, we can calculate an expected value for a metric that is targeted by both variants by looking at the counters used to calculate the values for those metrics in **RQ1.1**. For instance, for a metric like L1 Loads Miss-Rate we can note down the average counts of misses and hits in the Level-1 data cache, for either of the two variants (assuming both variants target the metric), across multiple executions of each. The expected value for that metric, in the context of a paired execution, is then obtained by:

$$\frac{Mean\_L1\_misses_{v_1} + Mean\_L1\_misses_{v_2}}{(Mean\_L1\_misses_{v_1} + Mean\_L1\_misses_{v_2}) + (Mean\_L1\_hits_{v_1} + Mean\_L1\_hits_{v_2})} \tag{5.3}$$

Overall, we obtain independent samples of values for each of the metrics that are targeted by both variants in a pair, across multiple combined executions. Then, we perform a one sample t-test for each such sample, using the calculated expected values as the theoretical means to test against. Naturally, we only perform the test when the normality assumption of the t-test is fulfilled - again, we rely on the Shapiro-Wilk method to assess this. Moreover, we take special interest in verifying whether there exist pairs of variants (v_1, v_2) that obtain significantly different impacts from the corresponding, reordered pair (v_2, v_1). Essentially, if our t-tests provide differing verdicts for such pairs, we perform additional verification using ANOVA [49], whereby we compare the samples of the reversely-ordered pairs against one other, provided that the normality assumption holds for the samples of each pair.

To validate the overall, weaving-based approach (**RQ2.2**), we design another experiment in which we can derive variants of our template configurable systems, wherein the same variant of the template can incorporate different combinations of gadget variants in its own structure. As such, we make full use of the decoupling between the template configurable system and the gadgets that the ISG facilitates. Specifically, we introduce a weaving mode in our Experiment component that allows us to get multiple ISGs for the same set of injection points, making use of uniform random sampling [5] to select a random group of gadget variants as injection candidates. Subsequently, we perform measurements for our template system, for each ISG, whereby we obtain multiple samples of values for several metrics. Note that in this case, we are interested in the *union* of metrics that are targeted by the injected gadget variants, since we want to evaluate every possible impact that can be enabled by any of the variants. The template system we use in this case is the Scrambler, introduced in Chapter 4. This template system comprises workloads with minimal impacts on resources, for each of its variants, which is useful to us for the purpose of having a real (no-injections) baseline against which we can compare the scenarios where we have multiple injections of gadgets for that same variant of the template.

5.3 RESULTS

In this section we provide the results of the experiments and measurements we described previously. We do so by focusing on each question and sub-question in turn. As such, we present technical details regarding sample sizes, statistical tests, hypotheses and any issues that we have encountered. These results will enable our discussion in the next section.

5.3.1 *Research Question 1*

5.3.1.1 *Consistency*

The results of our measurements with regard to **RQ1.1** are presented as follows. In Table 5.3 and Table 5.5 we present a statistical view of the IM_{ref}s of the Cache Gadget and Branch Prediction Gadget, respectively. We make use of all the metrics in the M_{cg} and M_{bpg} sets, previously described in Table 4.1 and Table 4.3. For each variant, we provide the mean values and CoVs for all metrics in the respective sets. Notably, we have opted not to omit values for the metrics that are not explicitly targeted by the gadget variants, though we have emphasized the targeted metrics with the color green. The size of each sample that we have used to calculate each mean and CoV value was set to 50.

Besides the values for the targeted metrics, we wanted to provide a better idea of what it means for a metric to not be targeted. Specifically, for non-targeted metrics we would ideally have minimal values of the mean, whereas the CoV can vary. We specifically see that this is the most often the case, though some exceptions exist which we highlight in red to indicate them as potential side-effects. In this context, we see side-effects as unintended cases in which some gadget variant obtains a high and/or stable impact w.r.t. a metric that it does not target by design. To further augment our results, we provide bootstrapped confidence-intervals on the basis of resampling, the confidence level being 90%. The bootstrapping is done by resampling 1000 times while also performing bias-correction-analysis [46], which is important when the samples do not resemble a normal distribution. These results are presented in Table 5.4 and Table 5.6, for each of the two gadgets.

One particular issue arises with regard to the Load Page-Walks and Store Page-Walks metrics in this case: we have omitted these metrics from the calculation due to there being multiple values in the original event counts that contained zero-values, which made the calculation of the confidence-intervals impossible. In addition, we have introduced some Not-Available (NA) entries for the L3 Loads Miss-Rate and L3 Store Miss-Rate for the same reason, and followed the same approach for Frontend-Boundedness and Bad-Speculation.

5.3.1.2 *Reproducibility*

Regarding **RQ1.2**, we have performed a multitude of Kruskal-Wallis tests, each involving four groups. Specifically, we have compared samples obtained for each variant-metric pair, in each of the four Intel environments listed in Table 5.2. The overall goal of these tests is to verify whether the samples indicate statistically significant differences in terms of the shapes of their respective frequency-distributions. Our choice of the Kruskal-Wallis was specifically informed by multiple goodness-of-fit tests that ruled out the possibility of using parametric methods such as ANOVA. As mentioned previously, we have used the Shapiro-Wilk test to check whether the samples obtained for the same variant-metric pair constitute normal distributions. The results for these normality tests are presented in Table 5.7 and Table 5.8. We have labeled each test environment using capital letters: S(kylake), K(abylake), H(aswell), B(roadwell).

The results of the Kruskal-Wallis tests themselves are presented separately, in Table 5.9 and Table 5.10. Again, we use the labels mentioned previously for the environments, and

Variant	L1 Load Miss-Rate		L2 Load Miss-Rate		L3 Loads Miss-Rate		L3 Store Miss-Rate		Load Page Walks		Store Page Walks		CPI	
	μ	CoV	μ	CoV	μ	CoV	μ	CoV	μ	CoV	μ	CoV	μ	CoV
v_1	99.66%	0.00	0.00%	0.46	4.11%	2.08	0.00%	0.00	5.27e+06	0.08	0.40	9.90	0.25	0.01
v_2	8.62%	0.35	99.99%	0.00	99.99%	0.00	52.30%	1.59	2.21e+10	0.27	11.92	2.41	17.10	0.27
v_3	99.99%	0.00	88.11%	0.07	0.00%	4.99	0.00%	0.00	7.97e+09	0.08	6.00	1.68	0.38	0.01
v_4	24.33%	0.00	94.49%	0.00	0.00%	0.40	0.00%	0.00	1.61e+10	0.01	1.50	8.66	0.38	0.01
v_5	99.96%	0.00	99.99%	0.00	80.69%	0.00	0.00%	0.00	1.78e+10	0.01	2.89	3.60	1.18	0.01
v_6	24.41%	0.01	99.99%	0.00	80.34%	0.00	0.00%	0.00	2.35e+10	0.00	55.40	1.18	0.93	0.00
v_7	0.00%	1.19	52.43%	0.66	0.97%	26.20	80.41%	0.01	2.55e+09	0.01	5.16e+07	0.01	1.19	0.09
v_8	0.00%	0.46	6.07%	1.50	0.00%	0.00	100.00%	0.04	3.21e+09	0.16	3.81	0.00	14.49	0.16
v_9	3.21%	0.58	32.88%	0.73	91.82%	0.17	99.99%	0.00	2.60e+10	0.01	1.03e+08	0.01	58.87	0.01

Table 5.3: Descriptive statistics (μ, CoV) for the IM_{ref} of the Cache Gadget. The green-shaded cells indicate targeted metrics, whereas the red-shaded cells indicate potential side-effects. Values are rounded to two decimal points.

Variant	L1 Load Miss-Rate		L2 Load Miss-Rate		L3 Loads Miss-Rate		L3 Store Miss-Rate		CPI	
	min	max	min	max	min	max	min	max	min	max
v_1	99.9664%	99.9668%	0.18e-03%	0.2e-03%	2.92%	5.87%	NA	NA	0.251	0.252
v_2	8.13%	9.13%	99.995%	99.999%	99.998%	99.999%	38.03%	65.82%	16.31	17.85
v_3	99.9993%	99.9994%	88.02%	88.23%	0.09e-06%	0.52e-06%	NA	NA	0.379	0.380
v_4	24.31%	24.34%	94.42%	94.56%	0.12e-05%	0.14e-05%	NA	NA	0.383	0.384
v_5	99.9681%	99.9687%	99.995%	99.999%	80.63%	80.74%	NA	NA	1.186	1.190
v_6	24.37%	24.45%	99.9963%	99.9964%	80.32%	80.36%	NA	NA	0.925	0.926
v_7	0.20e-03%	0.30e-03%	46.47%	57.90%	0.00%	4.99%	80.24%	80.62%	1.18	1.21
v_8	0.0010%	0.0012%	4.75%	7.85%	NA	NA	99.96%	100.00%	14.09	14.88
v_9	2.92%	3.54%	29.21%	37.03%	88.37%	93.96%	99.98%	100.00%	58.83	58.93

Table 5.4: Bootstrapped 90% Confidence-Intervals for the IM_{ref} of the Cache Gadget. Values are rounded to at least two decimal points, and to as many decimal points as needed to make the interval boundaries noticeable.

we report the p-values obtained from each respective test. As such, for each variant-metric pair we have used the following null hypothesis:

H_{k_0}: The concerned gadget variant has the same impact in each of the four subject environments, with regard to a specific metric.

Notably, we use a significance level of $p = 0.01$. The reason for this value is that we wanted to increase the possibility of retaining the null hypothesis even for values that are slightly smaller than 0.05, since we believe that rank-based comparisons are too sensitive, especially if the sample sizes are large or if they concern sub-decimal values. In addition, each of the samples used for these tests has a size of 30 values. Moreover, for this part we have opted to not report any p-values for metrics that are not targeted.

Variant	Branch Misprediction-Rate		Conditional Branch Misprediction-Rate		Front-end Boundedness		Bad Speculation		CPI	
	μ	CoV	μ	CoV	μ	CoV	μ	CoV	μ	CoV
v_1	13.21%	0.01	19.78%	0.01	53.94%	0.02	77.53%	0.01	2.22	0.13
v_2	11.10%	0.00	0.01%	1.01	95.50%	0.02	0.01%	1.01	0.79	0.01
v_3	13.16%	0.01	19.72%	0.01	51.63%	0.03	67.87%	0.01	1.75	0.01
v_4	11.15%	0.02	0.01%	0.32	81.78%	0.05	0.23%	4.62	0.65	0.03
v_5	9.09%	0.01	30.39%	0.06	5.43%	0.05	37.33%	0.05	266.71	0.00
v_6	15.07%	0.01	22.59%	0.01	75.88%	0.00	73.65%	0.00	2.31	0.01
v_7	9.48%	0.01	42.52%	0.01	99.09%	0.02	43.90%	0.01	1.35	0.01

Table 5.5: Descriptive statistics (μ, CoV) for the IM_{ref} of the Branch Prediction Gadget. The green-shaded cells indicate targeted metrics, whereas the red-shaded cells indicate potential side-effects.Values are rounded to two decimal points.

Variant	Branch Misprediction-Rate		Conditional Branch Misprediction-Rate		Front-end Boundedness		Bad Speculation		CPI	
	min	max	min	max	min	max	min	max	min	max
v_1	13.19%	13.25%	19.74%	19.82%	53.73%	54.15%	77.37%	77.62%	2.22	2.23
v_2	11.100%	11.104%	0.0068%	0.0070%	95.47%	95.53%	0.06%	0.08%	0.785	0.787
v_3	13.14%	13.17%	19.69%	19.73%	51.39%	51.85%	67.80%	67.91%	1.75	1.760
v_4	11.12%	11.19%	0.0057%	0.0063%	81.14%	82.48%	0.06%	0.41%	0.647	0.653
v_5	9.088%	9.091%	29.86%	30.57%	5.40%	5.50%	37.31%	37.35%	266.65	266.75
v_6	15.05%	15.09%	22.55%	22.62%	75.82%	75.93%	73.62%	73.67%	2.30	2.31
v_7	9.48%	9.50%	42.49%	42.55%	98.77%	99.29%	43.86%	43.93%	1.351	1.354

Table 5.6: Bootstrapped 90% Confidence-Intervals for the IM_{ref} of the Branch Prediction Gadget. Values are rounded to at least two decimal points, and to as many decimal points as needed to make the interval boundaries noticeable.

To add more context to our results, in Table 5.9 and Table 5.10 we have highlighted (in red) two cases when we observed a total loss of effect for a given variant-metric pair, in one of the environments. Similarly, we have highlighted (in yellow) several cases when the test would not reject the null hypothesis if the test were performed using only two environments, namely Skylake and Kabylake. For completion, we include descriptive statistics for the samples obtained in the three non-reference environments (Kabylake, Haswell, Broadwell), in Section A.2. Lastly, we provide separate results for the AMD environment listed in Table 5.2, in Table 5.11. For this environment, we have only included three metrics overall, due to differences in the semantics of the underlying counters that we would have to use to obtain these metrics, explained in Section 5.5.

Variant	L1 Load Miss-Rate				L2 Load Miss-Rate				L3 Load Miss-Rate				L3 Store Miss-Rate				CPI			
	S	K	H	B	S	K	H	B	S	K	H	B	S	K	H	B	S	K	H	B
v_1	✓		✓	✓							✓								✓	✓
v_2			✓																✓	✓
v_3							✓												✓	
v_4					✓						✓									
v_5							✓	✓	✓		✓	✓							✓	
v_6	✓		✓	✓				✓	✓		✓	✓							✓	
v_7									✓	✓	✓				✓	✓				✓
v_8			✓				✓		✓	✓					✓				✓	✓
v_9			✓	✓			✓	✓			✓	✓		✓	✓			✓	✓	✓

Table 5.7: Results for the normality tests regarding the IM_{ref} and other IMs of the Cache Gadget, shown for each of four test environments: S=Skylake, K=Kabylake, H=Haswell, B=Broadwell. Each checkmark indicates that the corresponding sample is likely obtained from a normally-distributed population.

Variant	Branch Misprediction-Rate				Conditional Branch Misprediction-Rate				CPI			
	S	K	H	B	S	K	H	B	S	K	H	B
v_1		✓	✓				✓				✓	
v_2	✓			✓	✓	✓					✓	✓
v_3							✓				✓	
v_4			✓			✓		✓			✓	✓
v_5	✓	✓	✓	✓		✓	✓	✓			✓	✓
v_6	✓	✓		✓	✓	✓		✓	✓	✓		✓
v_7			✓	✓			✓	✓			✓	✓

Table 5.8: Results for the normality tests regarding the IM_{ref} and other IMs of the Branch Prediction Gadget, shown for each of four test environments: S=Skylake, K=Kabylake, H=Haswell, B=Broadwell. Each checkmark indicates that the corresponding sample is likely obtained from a normally-distributed population.

Variant	L1 Load Miss-Rate	L2 Load Miss-Rate	L3 Loads Miss-Rate	L3 Store Miss-Rate	CPI
v_1	0.12	NT	NT	NT	NT
v_2	NT	0.82	$\ll 0.01$	NT	$\ll 0.01$
v_3	0.03	$\ll 0.01$	NT	NT	NT
v_4	$\ll 0.01$	$\ll 0.01$	NT	NT	NT
v_5	$\ll 0.01$	$\ll 0.01$	$\ll 0.01$	NT	NT
v_6	$\ll 0.01$	$\ll 0.01$	$\ll 0.01$	NT	NT
v_7	NT	NT	NT	$\ll 0.01$	NT
v_8	NT	NT	NT	0.52	$\ll 0.01$
v_9	NT	NT	$\ll 0.01$	$\ll 0.01$	$\ll 0.01$

Table 5.9: Results of the Kruskal-Wallis tests regarding the Cache Gadget, obtained for the four Intel environments. Green-shaded cells indicate cases when the H_{k_0} would be retained. Yellow-shaded cells indicate cases when the hypothesis is rejected but would be retained if we had used only the Skylake and Kabylake environments in the test. Red-shaded cells indicate cases when a total loss of impact was observed in one of the four environments. NT=Non-Targeted.

Variant	Branch Misprediction-Rate	Conditional Branch Misprediction-Rate	CPI
v_1	$\ll 0.01$	$\ll 0.01$	NT
v_2	$\ll 0.01$	NT	NT
v_3	$\ll 0.01$	$\ll 0.01$	NT
v_4	$\ll 0.01$	NT	NT
v_5	$\ll 0.01$	$\ll 0.01$	$\ll 0.01$
v_6	0.13	$\ll 0.01$	NT
v_7	$\ll 0.01$	$\ll 0.01$	NT

Table 5.10: Results of the Kruskal-Wallis tests regarding the Branch Prediction Gadget, obtained for the four Intel environments. Green-shaded cells indicate cases when the H_{k_0} would be retained. Yellow-shaded cells indicate cases when the hypothesis is rejected but would be retained if we had used only the Skylake and Kabylake environments in the test. Red-shaded cells indicate cases when a total loss of impact was observed in one of the four environments. NT=Non-Targeted.

Gadget	Variant	Branch Misprediction-Rate	Conditional Branch Misprediction-Rate	L1 Load Miss-Rate	CPI
BP Gadget	v_1	0.44%	0.00%	NA	0.46
	v_2	1.89%	0.00%	NA	0.36
	v_3	0.04%	0.00%	NA	0.37
	v_4	1.39%	0.00%	NA	0.29
	v_5	8.97%	0.00%	NA	9.06
	v_6	0.00%	0.00%	NA	0.41
	v_7	4.90%	0.00%	NA	1.13
Cache Gadget	v_1	NA	NA	25.08%	0.36
	v_2	NA	NA	NT	41.0
	v_3	NA	NA	20.36%	0.25
	v_4	NA	NA	2.28%	0.18
	v_5	NA	NA	61.45%	1.93
	v_6	NA	NA	9.09%	1.03
	v_7	NA	NA	NT	1.09
	v_8	NA	NA	NT	47.93
	v_9	NA	NA	NT	3.26

Table 5.11: Partial results regarding the IM of each gadget, in the AMD Zen3 environment. NA=Not-Available. NT=Non-targeted.

5.3.2 *Research Question 2*

5.3.2.1 *Inter-Gadget Influences*

Regarding **RQ2.1**, we have conducted an experiment as outlined in Section 5.2.2. Considering that we have performed a multitude of one-sample t-tests, the results we present here are aggregated and organized in a breakdown of important findings, shown in Table 5.12. Each test concerns a sample of values obtained for a metric that is targeted by both gadget variants in a given pair, with the sample size being set to 20 and the significance level being $p = 0.01$. Overall, our tests rely on the following null hypothesis and one-sided alternative hypothesis:

H_{t_0}: The mean of a sample of values obtained for a metric that is targeted by both gadget variants in a combined execution is the same as the expected value we manually calculate.

H_{t_a}: The mean of a sample of values obtained for a metric that is targeted by both gadget variants in a combined execution is **less** than the expected value we manually calculate.

One key aspect that we aimed to show in Table 5.12 is that there are indeed cases when, for some pairs of gadget variants, the means of the collected samples of values for some

Gadget	Pair Type	Retains		Rejects		Non-Normal
		#	δ_μ	#	δ_μ	#
BP Gadget	All	36	0.08%	14	0.29%	24
	Same Variant	7	0.14%	1	0.16%	4
Cache Gadget	All	45	0.30%	17	1.35%	48
	Same Variant	7	0.06%	3	3.39%	10

Table 5.12: The breakdown of our one-sample t-tests regarding our experiment with manually defined expected values. # indicates counts, whereas δ_μ indicates the mean of the differences obtained by subtracting the manually defined expected-value from the mean of each sample for which we perform a test, and then taking the average value of these differences. The δ_μ is a percentage because the concerned metrics are also percentages (e.g. L2 Miss-Rate).

metrics do not meet the respective expected values that were defined by us. In addition, we distinguish between cases when the pairs constitute two identical variants as opposed to when they include different variants (of the same gadget). In that regard, we note that identical-variant pairs are less prone to rejecting H_{t_0}.

Overall, the number of tests we performed is 112, out of which 31 resulted in H_{t_0} being rejected in favor of H_{t_a}. We can also see that the Cache Gadget is more prone to rejections of the null hypothesis. To further contextualize our results, we provide a statistic we call the *mean* of the differences: δ_μ. Firstly, differences are calculated by subtracting the manually-defined expected value from the mean of each sample of values. Then, the mean of the differences is simply the average value of all the calculated differences. This statistic should provide a rough idea of how much these values deviate from the expected value. Indeed, we see that the values of δ_μ are larger for the cases when the null hypothesis is rejected.

Besides the panel of one-sample t-tests, we also report cases when the sample corresponding to some pair (v_1, v_2), presents a different verdict for H_{t_0} than its reordered counterpart (v_2, v_1), with regard to the same metric. These cases are summarized in Table 5.13. For each such case, we have performed additional verification by taking the samples of metric values for each of the pairs and comparing them using ANOVA (cf. Section 5.2.2), with the null hypothesis being as follows:

H_{a_0}: The means of the two samples of values obtained for a pair and its reordered counterpart with regard to a specific metric, are the same.

Our results indicate that all the disagreeing pairs we identified based on our t-tests, also reject H_{a_0}. We provide the respective p-values in Table 5.13 (again, using a significance level of $p = 0.01$), where we also name the concerned metric.

5.3.2.2 *Weaving*

As the last part of our results, we present our findings regarding the use of injection points in a given template configurable system (**RQ2.2**). The results are shown in Table 5.14. The table shows different combinations of variants that were generated by uniformly and randomly

Gadget	Pair	p-value	Culprit
BP Gadget	(v_1, v_6)	2.09e-13	Branch Misprediction-Rate
	(v_2, v_6)	1.28e-57	Branch Misprediction-Rate
	(v_2, v_7)	2.08e-53	Branch Misprediction-Rate
	(v_3, v_6)	2.40e-16	Branch Misprediction-Rate
	(v_4, v_6)	2.15e-24	Branch Misprediction-Rate
	(v_5, v_7)	7.45e-63	Branch Misprediction-Rate
	(v_6, v_7)	2.72e-45	Conditional Branch Misprediction-Rate
Cache Gadget	(v_3, v_5)	6.30e-38	L2 Load Miss-Rate
	(v_5, v_6)	2.70e-07	L3 Load Miss-Rate
	(v_6, v_9)	2.39e-07	L3 Load Miss-Rate

Table 5.13: The results for the ANOVA-based verification regarding cases when pairs provide different effects than their reordered counterparts.

sampling the set of gadget variants listed in Table 5.1, and how the impact of a specific variant of our Scrambler template system fares when these gadget variants are injected in some statically-defined locations in the code of the template system that corresponds to the features of that (template) variant. This effectively provides us with multiple ISGs and consequently multiple synthetic benchmarks. The table also shows how the selected variant of the template fares when no gadget variants are injected, which provides us with a baseline view of the template's variant itself.

For each ISG we obtain samples of metric values of size 20. We focus mostly on the metrics that are targeted by *at least* one of the gadget variants participating in the ISG, as explained in Section 5.2.2. As such, we highlight these metrics with either green or yellow, whereas values for non-targeted metrics are also included. The green cells indicate cases when, compared to the baseline, the ISG has been effective in altering the performance profile of the template variant. The yellow cells indicate the same thing, but with the small difference that the alterations of the impacts are relatively small compared to the baseline. Lastly, the red cells indicate non-targeted metrics for which we saw unexpectedly high impacts - this echoes the potential side-effects idea mentioned previously for **RQ1.1**.

Gadget Variants	Branch Misprediction-Rate	Cond. Branch Misprediction-Rate	L1 Load Miss-Rate	L2 Load Miss-Rate	L3 Load Miss-Rate	L3 Store Miss-Rate	CPI
Baseline	0.00%	0.00%	0.00%	16.39%	13.41%	38.17%	0.44
$(CacheGadget, v_7)$, $(CacheGadget, v_5)$, $(BPGadget, v_2)$	10.00%	0.00%	1.48%	96.33%	78.69%	80.00%	0.85
$(CacheGadget, v_9)$, $(CacheGadget, v_8)$, $(CacheGadget, v_2)$	0.06%	0.00%	3.08%	99.45%	99.98%	99.89%	15.60
$(BPGadget, v_5)$, $(BPGadget, v_4)$, $(BPGadget, v_7)$	14.02%	20.09%	0.38%	72.76%	97.41%	12.45%	6.03
$(CacheGadget, v_4)$, $(CacheGadget, v_6)$, $(BPGadget, v_2)$	7.29%	0.00%	3.80%	93.08%	28.25%	2.18%	0.71
$(BPGadget, v_4)$, $(BPGadget, v_6)$, $(BPGadget, v_3)$	13.94%	20.72%	0.17%	5.89%	2.69%	30.83%	1.99%
$(CacheGadget, v_5)$, $(CacheGadget, v_1)$, $(CacheGadget, v_6)$	0.00%	0.00%	49.89%	49.58%	80.49%	20.47%	0.75
$(CacheGadget, v_2)$, $(BPGadget, v_7)$, $(CacheGadget, v_6)$	2.98%	6.20%	11.85%	99.99%	82.53%	89.03%	1.68
$(BPGadget, v_3)$, $(BPGadget, v_3)$, $(BPGadget, v_4)$	9.16%	12.23%	0.19%	5.98%	0.29%	3.03%	0.95
$(CacheGadget, v_4)$, $(CacheGadget, v_9)$, $(BPGadget, v_6)$	7.26%	11.45%	9.08%	91.74%	0.00%	100.00%	1.31
$(BPGadget, v_3)$, $(CacheGadget, v_4)$, $(CacheGadget, v_1)$	0.42%	0.73%	32.88%	45.09%	0.00%	20.52%	0.36

Table 5.14: Results on several *benchmarks*, generated on the basis of multiple ISGs that are obtained using uniform random sampling of all gadget variants, and a single variant of a template system. The green-shaded cells indicate cases when the metric is targeted by at least one of the injected gadget variants and the measured impact is significantly higher than the baseline. Yellow-shaded cells indicate cases when the metric is targeted by at least one injected gadget variant but the measured impact is only slightly higher than the baseline. The red-shaded cells indicate non-targeted metrics for which we see a potential side-effect in terms of the measured impact.

5.4 DISCUSSION

In this section we interpret the results we presented previously. Our interpretation is focused on directly answering the two research questions that we introduced at the beginning of the chapter, though we also try to put things in context by pointing out how our findings relate to the overall approach, and ultimately to the goals that we outlined early in the thesis.

5.4.1 *Research Question 1*

The results presented with regard to **RQ1.1** provide sufficient evidence in favour of the gadgets being highly consistent. This is clearly visible by examining the reported values of CoV, for either of the two gadgets. In fact, we do not observe a single case when the CoV is

above 0.5, which we consider to be a threshold that would indicate a significant dispersion of values around the mean in our context. The consistency of gadgets can be further seen by looking at the bootstrapped confidence-intervals. All the reported intervals are very narrow, with the overall ranges being sub-decimal in relation to the interval boundaries. We believe that this level of stability is a direct result of our resource-focused approach through which we were able to tune the impact of each gadget variant.

Turning our attention to the second part of **RQ1**, we notice that there is a challenge in being able to carry the highly consistent impacts measured in our reference environment, to other environments. This is evidenced by the results of our cross-environment normality and Kruskal-Wallis tests, which indicate that there is, in most cases, a statistically significant difference in the values we obtain across four environments. This is the case even though the four environments we chose represent subsequent generations of Intel processors. As such, this clearly validates our suspicion regarding microarchitectural differences being too important to ignore. The problem is further noticed when looking at the results obtained in the AMD Zen3 environment, where we see many gadget variants losing their impacts entirely, although that is something we had anticipated given the fact that we focused mostly on Intel-based literature during implementation.

Overall, we cannot say that the gadgets provide **strictly** reproducible behaviors across different environments. However, based on the metric values we have obtained for each of the four Intel environments, we see that gadget variants obtain large impacts that, for practical purposes, can allow researchers to rely on our gadgets in these environments. To better address the issue of reproducibility, we present an additional perspective in Figure 5.1.

Based on the figure, we observe that the shapes of the frequency distributions for the values we obtain (w.r.t. a specific gadget variant and a specific metric) differ significantly between two environments, even though the impacts remain large in both. In the context of Kruskal-Wallis, this is important because the test typically picks up on differences in terms of the shapes of distributions. For instance, it observes how the data is spread around the median, and our violin plot clearly indicates that the spread of data (violin width) around the median (white dot) can vary across environments. If the shapes of distributions are deemed largely similar, the test then simply compares the median values to decide whether the two distributions are significantly different.

In addition to the shape differences, we also see that the interquartile-ranges can also differ significantly, as indicated by the embedded black box-plots. Moreover, besides the overall shapes, the two distributions do not overlap in terms of their values and thus have different medians, though the differences are relatively small for practical purposes.

Regardless, in spite of not being able to claim reproducibility of behaviors across environments, we can at least confirm that our gadgets are practically useful in at least four Intel microarchitectures[4], considering that they obtain large and consistent impacts in each of them. This idea is particularly important regarding our definition of GT - based on these results, we can only define the GT of our gadgets experimentally, unless truly microarchitecure-independent techniques are identified and utilized in our gadgets (cf. Chapter 7).

4 Please refer to Section A.2 for the aggregated data for Haswell, Broadwell and Kabylake

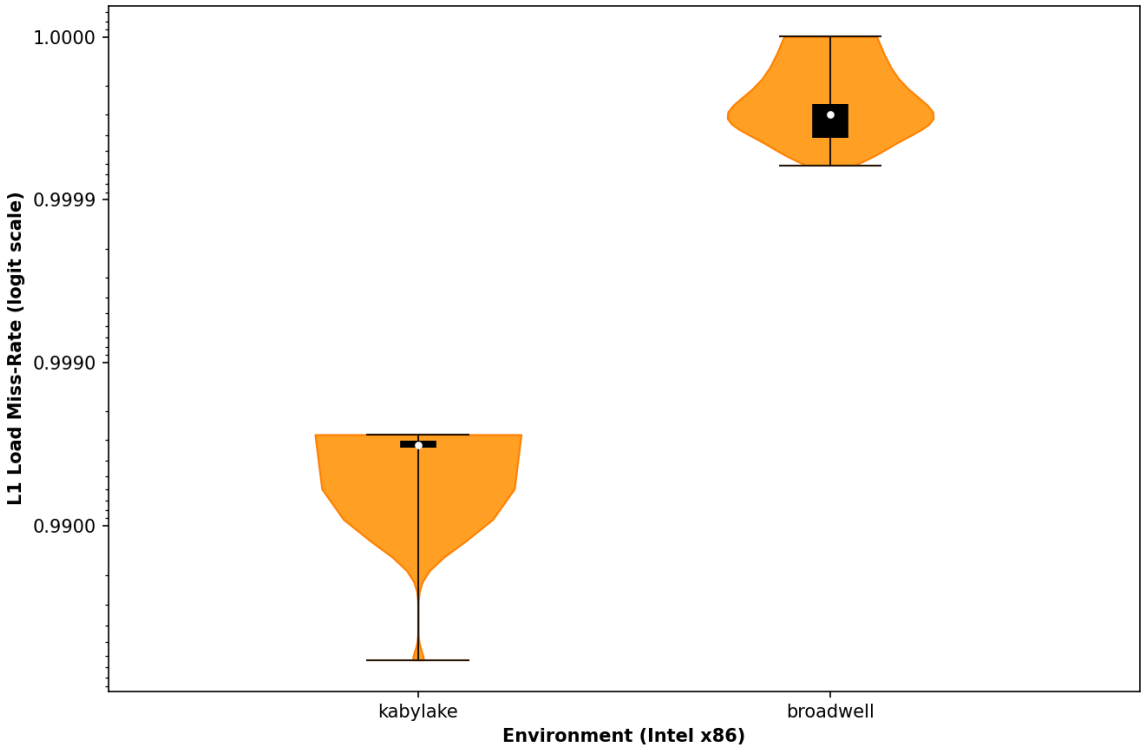

Figure 5.1: Violin plots, with embedded box-plots, for two of the four Intel environments. The violins are constructed using Kernel-Density Estimation (KDE), with a density parameter of 20. The white-dot in each box-plot indicates the median. Each violin corresponds to a sample of values of L1 Load Miss-Rate for v_1 of the Cache Gadget.

Research Question 1 - Verdict:
The gadgets provide us with highly consistent behaviors across multiple executions of their variants in the same environment, but the exact reproducibility of these behaviors can not be guaranteed for arbitrary environments. In particular, our results indicate that we can only define the ground-truths of gadgets by experimentally obtaining impact-mappings in each environment of interest.

5.4.2 *Research Question 2*

with regard to **RQ2**, our first experiment helped us confirm that there can be cases when two variants influence one-another. This finding confirms our suspicion regarding potential inter-gadget effects due to the way caching and branch-prediction works, though further research is needed to explicate the precise reasons for such effects. Furthermore, we have also found that there can be ordering effects, as evidenced by our ANOVA tests, which adds a dimension to our work that we had not foreseen. Such ordering effects can complicate the design of benchmarks, though the large number of potential ISGs we can have for the same template variant implies that it is possible to obtain benchmarks that will not suffer from such effects. In any case, we aim to address inter-gadget influences in future-work (cf. Chapter 7).

Regardless of the discovery of potential inter-gadget influences, we have been able to verify that, relative to a given baseline, gadget injections can effectively alter the behavioral profile of an existing system. This provides us with a clear path towards, for instance, synthesis of outsized impacts in cases when two features of some configurable (template) system interact structurally - indeed, that is simply a matter of placing the injection points in locations where "glue" code exists to facilitate such an interaction between two features, as shown in Listing 5.1. This clearly validates the usefulness of our weaving approach in synthesizing programs with properties and behaviors of interest. Having put this approach in place, all that is left is to build templates with predefined injections such as the ones shown in the code snippet, which would allow us to obtain Software Product Lines (SPLs) of benchmarks as explained in Chapter 3. We have already done this for two miniature templates (cf. Chapter 4), but we did not explicitly dive into specifics here because our focus was on validating the overall approach.

```
#ifdef MULTPLICATION
    void config_mul(layout* l, representation r) {...}
#endif
.
.
void setup(layout* l){
    .
    r = get_default_representation();
#ifdef MULTIPLICATION
#ifdef SCIENTIFIC
    //Interaction with manipulated profile
    #IP_1
    #IP_2
    #IP_3
    r = get_binary_representation();
#endif
    config_mul(l, r);
#endif
    .
}
```

Listing 5.1: The feature-interaction example introduced in Chapter 2, but now with injection points. Injecting gadget variants at these points will make the feature-interaction have a distinct profile w.r.t. the metrics that the injected gadgets target.

On a less positive note, an important consideration w.r.t. our results is the fact that there might be "side-effects" in the behavior of some gadget variants. Such side-effects can be noticed both when the variants are used in isolation, and more importantly, when they are injected in a template system. Moreover, these effects can potentially happen in two ways. Firstly, a gadget variant can have a larger-than-intended impact concerning a metric that it does not explicitly target. Secondly, a gadget variant can obtain sizeable impacts in terms of metrics that the gadget does not include in its definition, but which are included in the definition of the other gadget. Either of the two cases can manifest in and affect the stability

of the behaviors that the resulting benchmarks have. As such, further work is needed to address this lacking aspect of our approach (Chapter 7).

Research Question 2 - Verdict:
Our approach is effective in producing synthetic benchmarks on the basis of gadgets, although potential issues might exist for some gadget variants or combinations thereof, which can reduce their individual contributions to the behaviors of these benchmarks.

5.5 THREATS TO VALIDITY

Our results can be impacted by some internal biases in how we perform measurements and experiments. Therefore, we identify several potential threats to the validity of our findings, both internal and external. In terms of internal threats, we observe that our measurements for **RQ1.1** could be impacted by some level of carry-over effects in between executions, which can amplify the values we observe overall. Furthermore, the choice of the reference environment was arbitrary; the approach we followed was such that during the implementation phase we needed to tune and specify the impact of our gadgets through trial-and-error. This might have led us to tailor our gadgets too much to that environment, although we tried to implement gadgets using techniques that we deemed to be, to some extent, microarchitecture-independent.

Internal threats could exist w.r.t. our experiment in **RQ2.1**, which can suffer from bias because the way we defined the *expected* values depended on the samples we had obtained for **RQ1.1**, when using the gadgets in isolation. As such, any sampling errors could impact our expected values. Moreover, the choice of the variant of the Scrambler template in **RQ2.2** could present a case in which the baseline profile is too "mild" in terms of the concerned metrics, which makes it easy for the injected gadgets to alter the impact of the template variant. Regardless, we believe that the random sampling idea we used in picking different gadget variants for injections shows that most gadget variants will be highly successful, which gives us confidence w.r.t. other template variants.

Other important threats we identify are of an external nature. Firstly, we acknowledge that the choice of our environments in **RQ1.2** is not ideal for assessing reproducibility. Indeed, we would need to have larger variety in the environments we choose by including more AMD systems and newer Intel ones. This became obvious to us during experimentation as we struggled with the semantics of some PMU counters, thus realizing the gravity of microarchitectural differences among environments. In particular, obtaining values for a metric like L1 Loads Miss-Rate requires us to obtain counts for the misses and hits that occur in that level of caching; however, in Intel systems, the available counters typically count prefetching-induced references, whereas in AMD they not always do. In addition, in Intel systems several cache-misses can be buffered between the first two levels of caching and are thus combined into single events (from the perspective of counting), whereas for AMD this is not transparent. These reasons forced us to include only partial results for the fifth environment in Table 5.2. Moreover, for some variants of our gadgets we have relied on the use of intrinsic instructions offered in x86 systems, which make the evaluation of our work impossible in other types of microarchitectures. Regardless, choosing subsequent

generations of Intel x86 microarchitectures helps us ensure that we have a set of similar environments in which our gadgets work relatively well.

Lastly, we also acknowledge that our choice of the Scrambler template system for **RQ2.2** might make the results for that question less generalizable, although we have previously acknowledged that injecting gadgets in arbitrary systems is something we should pursue in the future.

6 RELATED WORK

6.1 COMPUTER ARCHITECTURE

The benchmarks we implemented in this thesis required a solid understanding of a system's hardware and logical components. To gain a better understanding of these components we relied on the seminal work of Drepper [19] which scoped several potential performance improvements that can be performed in high-level code, focusing specifically on memory units and subsystems. Furthermore, in [23, 24, 30, 50] we found details relating to locality ideas and prefetching strategies employed by the caches and inherent limitations. We also relied on the works of Denning [24], Smith and Sohi [31], Fog [33], and Seznec [45] to obtain a better understanding of the limitations that exist in current superscalar architectures. Notably, Fog compiled a microbenchmark-based survey of branch-prediction behaviors for Intel systems. Based on our survey of all these works, we were able to make use of many of their findings to produce performance regressions in our designs. We also relied on official manuals such as [20], especially regarding the use of intrinsic x86 instructions that help in bypassing caches, and details regarding the Performance-Monitoring Unit (PMU). From a performance-monitoring perspective, we followed the methodology outlined in [28], and the findings of Ameller et al. [37] and Berger and Guo [51] to perform and interpret measurements.

6.2 CONFIGURABLE SOFTWARE SYSTEMS

In order to develop a synthesis process for our synthetic benchmarks, we relied on notions of systematic reuse found in Software Product Line (SPL) literature. We initially turned our attention to literature surveys such as [13, 14], and the field-defining work on Feature-Oriented Software Development (FOSD) by Apel and Kästner [15]. These works enabled us to get a better understanding of the best practices in the field. Specifically, Svahnberg, Van Gurp, and Bosch [13] and Galster et al. [14] examined the semantics of SPL and collected information on current practices and limitations across tens of other published works. Furthermore, the work of Apel and Kästner [15] structured the discussion around feature-orientation in software engineering, which helped our work directly. The works of Umar and Khan [36], Ameller et al. [37], Fadul [38], and Berger and Guo [51] then helped us make a conceptual transition from non-functional properties to low-level metrics in our approach. From a more practical standpoint, the ideas presented by Schulze and Fenske [52] informed us on the inherent limitations of compile-time configurability techniques such as the use of preprocessor directives.

6.3 BENCHMARKS AND FAULT INJECTION

Our benchmarking concept stands on a fault-line between benchmarking systems and fault-injection frameworks. Ziade, Ayoubi, Velazco, et al. [41] created a taxonomy of the types of faults that software systems experience, and gave credence to the notion of performance faults. In addition, Natella et al. [42] provided a more focused look at the semantics of a fault, highlighting the importance of specifying not only the type, but also the in-code location of a faulty code segment, in order for fault-injection to be reliable. Based on these notions, we developed the idea of injected code-fragments that map to variants of a configurable system and represent tunable performance faults. Furthermore, the early work on microbenchmarks presented by McVoy, Staelin, et al. [10] and Kopytov [12] provided us with an initial understanding of synthetic benchmarks. These benchmarking systems constitute largely representative workloads that can help identify bottlenecks in the system, especially regarding memory considerations. However, they fall short in regards to achieving maximal "stress" on a target system. The notable works of Joshi et al. [9] and Van Ertvelde and Eeckhout [53] introduced us to more advanced benchmarking concepts such as statistical simulation, performance cloning and microarchitectural-independence considerations when constructing synthetic workloads. Both these works aimed to replicate real-world systems through a set of predefined code-fragments that supposedly apply microarchitecutre-independent workloads. Moreover, these code-fragments were typically weaved and combined through statistically generated flow-graphs, with the aim of replicating (cloning) the performance of a given real-world system. As such, our approach has major similarities but also crucial differences to these works.

Firstly, our solution incorporates code-units (gadgets) that are more coarse-grained and not microarchitecture-independent per se, but also configurable and hence tunable. Furthermore, we do not incorporate any sophisticated techniques for the mixing of these units; instead we focus on achieving specific goals related to software analysis by manually crafting execution templates, and provide a framework that allows the user to inject these units in a systematic and flexible manner in these templates. Therefore, we are able to fully decouple the design of these units from the synthesis process. On the other hand, our gadgets are not truly microarchitecture-independent in the sense that they do not have precisely reproducible impacts across different microarchitectures, which is something that the aforementioned works typically try and manage to address to a considerable extent. Nevertheless, we believe that our approach fully lends itself to automated synthesis as in the aforementioned works, as well as improvements in terms of microarchitecture-independence ideas. To be precise, while our approach constitutes neither *statistical simulation* nor *performance cloning*, it can serve as a foundation for novel approaches of this kind (cf. Chapter 7).

On similar notes, Van Ertvelde and Eeckhout [53] utilized execution traces from existing programs to automatically extract code-fragments that can then be re-weaved into simulated traces which, if executed, would produce similar performance profiles to the original program but with much less code. The extraction process is based on predefined predicates that largely resemble the programming techniques we used in our gadgets.

7 CONCLUSION

In this section, we provide a summary of the thesis goal and our findings. To further improve our contribution, we also outline potential future work directions based on ideas that we encountered at different stages of our work.

7.1 FUTURE WORK

7.1.1 *Potential Improvements*

Some important aspects of this work that we intend to address in future work relate to the side-effects and ordering-effects that we identified in Chapter 5. As a first step, we would need to come up with a precise definition of side-effects; this is not trivial because for a given workload of a gadget variant, we need to have a very intimate understanding of microarchitectures to be able to say whether some resource is impacted unduly or not. This issue becomes especially important when we consider that our benchmarks can use gadgets of different kinds; naturally, we would want to have gadgets behave in a mutually-exclusive way so that the behavioral profiles of the resulting benchmarks are easier to understand and interpret. Besides side-effects, we were able to identify that the execution order of gadget variants, in the context of a template system, could present some challenges. Therefore, we intend to verify this aspect in more detail, potentially by experimenting with a much larger number of gadget variants.

Additionally, there are two aspects that we wish to study further with regard to the reproducibility of the impacts of gadgets, across different microarchitectures. The first part is that we would like to make our gadgets less Intel-specific, potentially by drawing new programmatic techniques out of AMD-specific literature and manuals. This also implies that our resource-focused approach must be made more generic by accounting for AMD-specific architectural resources and performance counters. Secondly, we believe that the knobs and features of each gadget can be designed with the purpose of overcoming microarchitectural differences. For instance, we can have features that can be enabled only in specific environments, and which make use of some intricate aspect of that environment to achieve a sizeable impact. Similarly, we can potentially design and implement knobs that adapt the applied workloads to different environments, so as to make our gadgets work in a more microarchitecture-independent manner.

Lastly, one facet that can be added to our work on the basis of gadgets is that of incorporating multi-threaded techniques. This is interesting to pursue especially in the context of the Cache Gadget, where the coherency-protocols applied by the Central Processing Unit (CPU) in relation to the caches could be directly subjected to synthetic workloads.

7.1.2 *New Avenues*

From a broader perspective, our work on developing the Software Product Line (SPL) approach that was presented in Chapter 3 and detailed in Chapter 4 can potentially lead to improvements that make the overall weaving framework more efficient and helpful for researchers. For instance, we can use reconfigurability ideas [13, 54]. Specifically, we can conceptualize a weaving process that is performed while a given template system is executing. This would allow researchers to dynamically alter the behavioral profile of a program. A technique that could enable this is the use of user-defined tracepoints [28], which could potentially allow us to modify the execution trace of a program at runtime. Alternatively, we could transpose our framework into a simulation framework; for instance, we could be able to maintain some state for a running configurable system, based on which we could dynamically interrupt the execution of that system, re-inject desired gadget variants and resume execution. Having dynamic capabilities of this kind can prove useful when analysing programs that rely on runtime-configurability, or when trying to simulate and/or detect intermittent (short-lived) performance faults.

Another compelling direction is that of performing proper fault-injections. In our approach we made use of notions of injections, albeit to a minimal extent. Potentially, we could improve on the design of the gadgets and try to alter the behavioral profile of large, real-world systems. This would also require a proper taxonomy for different types of faults, as well as some semantic considerations regarding the delineation of faulty vs. non-faulty behaviors. In any case, the use of a modular component akin to our gadget can prove helpful in inducing abrupt impacts in a given existing system.

Lastly, we observe that the overall contribution of this thesis is the laying of groundwork towards configurability-based synthesis of workloads. In our case, the synthesis itself is largely static, since the injection points are predefined for each template system. A potential improvement in that regard is to define injection-points dynamically, and perhaps have some statistical input as to where the injection points should be placed. This would bring our implementation closer to the statistical simulation ideas discussed in Chapter 6, and therefore enhance the capability of getting fine-tuned, synthetic behaviors.

7.2 SUMMARY

In this work, we set out to provide a novel way through which researchers can generate synthetic benchmarks that help them in validating and improving their software analysis methods, in the context of configurable software systems. To achieve this, we followed a resource-focused approach. This approach then guided us to programmatic techniques through which we could design programs with stable, and consequently transparent behaviors, w.r.t. computer systems' resources. In addition, we relied on Feature-Oriented Software Development (FOSD) concepts to collate these techniques into feature-oriented designs that we call gadgets. A direct consequence of a gadget's feature-oriented design is the variety we can obtain in terms of synthetic workloads, while reusing the same code artifacts. Based on such notions of reuse, we then developed and implemented a two-level SPL concept. To realize this concept in practice, we crafted a framework that produces synthetic benchmarks by injecting variants of gadgets in the source-code of

existing, configurable software systems that act as execution templates, and thus provide a way of mixing the workloads that the gadgets entail. Based on this framework, and the gadgets' properties, we can construct benchmarks with varied, stable and transparent behaviors. To that end, in our evaluation we focused heavily on the gadget concept and scrutinized multiple aspects thereof. Notably, we assessed the stability of the gadgets' behaviors in at least four Intel systems. Moreover, we validated the usefulness of our approach as a whole, and showed that it provides researchers with the necessary ingredients for generating meaningful synthetic benchmarks.

APPENDIX

A.1 TEMPLATE FEATURE-MODELS

A.1.1 *The Scrambler template*

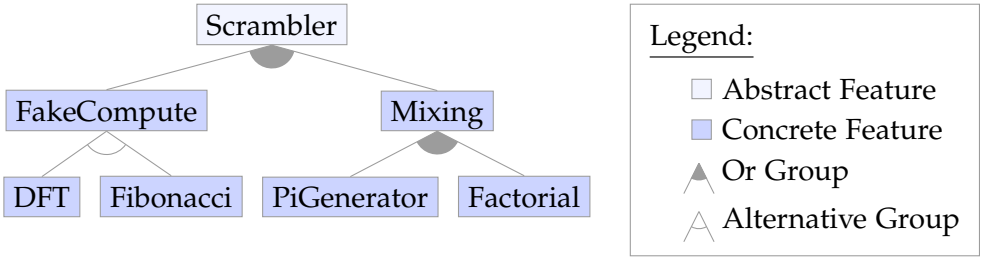

Figure A.1: Complete feature-model for the Scrambler template. DFT and Fibonacci provide series of computations, in tandem with the scrambling that is generated from the Mixing elements. For instance, Fibonacci uses a recursive implementation that generates the corresponding series, and uses the elements in the series to generate the numbers of π using a Taylor series formula. Alternatively, a Factorial can be taken for each element in the Fibonacci series.

A.1.2 *The Memo template*

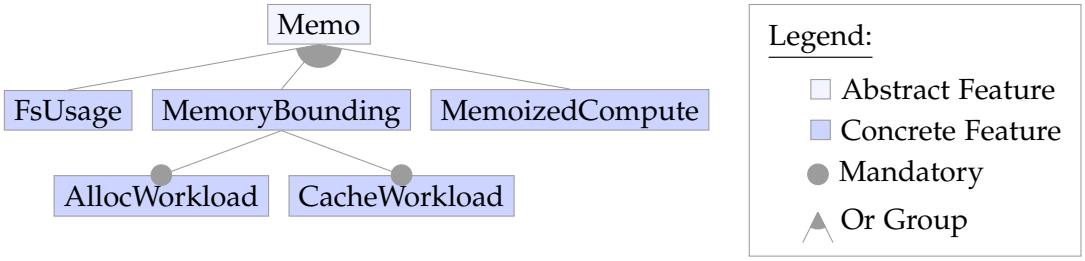

Figure A.2: Complete feature-model for the Memo template. FsUsage indicates a functionality that performs file operations. Memoized Compute refers to a memoized implementation of a simple computation task. AllocWorkload and CacheWorkload perform close-to-optimal memory allocations and cache acceses.

A.2 REPRODUCIBILITY RESULTS

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate	load-page-walks	store-page-walks
v1	0.2514	99.25%	0.63%	11.09%	0.00%	5.28E+09	0.0000
v2	19.9151	7.92%	100.00%	100.00%	82.97%	2.57E+10	2.0100
v3	0.3535	100.00%	95.40%	0.00%	0.00%	7.43E+09	0.0000
v4	0.3579	24.09%	91.16%	0.00%	42.30%	1.51E+10	8.8800
v5	1.4368	99.94%	100.00%	91.09%	8.69%	2.16E+10	1.0600
v6	1.1128	24.36%	99.99%	91.07%	21.83%	2.82E+10	262.1700
v7	1.1140	0.00%	39.47%	23.29%	91.72%	2.40E+09	52318557.1400
v8	16.9594	0.00%	24.30%	-12.11%	99.94%	3.75E+09	0.0000
v9	77.1191	0.32%	59.81%	93.94%	99.97%	3.40E+10	105513742.4700

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	cpi
v1	24.30%	16.20%	2.406
v2	0.00%	0.00%	0.392
v3	19.90%	13.30%	1.798
v4	0.00%	0.00%	0.337
v5	49.90%	9.10%	485.04
v6	22.30%	14.80%	2.222
v7	41.80%	9.30%	1.349

Figure A.3: Mean values for the gadget variants, in the Kabylake environment.

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate	load-page-walks
v1	0.2619	100.00%	0.00%	14.18%	0.00%	5.50E+09
v2	6.1857	4.53%	100.00%	99.99%	0.81%	8.00E+09
v3	0.3534	100.00%	100.00%	0.00%	0.00%	7.43E+09
v4	0.3815	24.84%	100.00%	0.00%	0.00%	1.60E+10
v5	1.3232	99.94%	99.99%	80.64%	0.00%	1.99E+10
v6	1.1100	24.40%	100.00%	80.29%	0.00%	2.84E+10
v7	1.3456	0.10%	83.79%	2.11%	80.68%	2.71E+09
v8	13.8485	0.00%	82.50%	77.88%	100.00%	3.06E+09
v9	56.1814	2.54%	99.58%	83.88%	81.14%	2.36E+10

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	cpi
v1	21.40%	12.20%	1.91
v2	0.00%	11.10%	0.841
v3	22.60%	12.90%	1.634
v4	0.00%	11.10%	0.658
v5	58.30%	9.10%	201.377
v6	27.00%	15.40%	2.222
v7	52.30%	10.00%	1.525

Figure A.4: Mean values for the gadget variants, in the Haswell environment.

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate
v1	0.2619	100.00%	0.00%	4.22%	0.00%
v2	15.5974	5.30%	99.99%	100.00%	0.78%
v3	0.4211	100.00%	94.68%	0.00%	0.00%
v4	0.4312	23.45%	93.90%	0.00%	0.00%
v5	1.6258	99.97%	100.00%	100.00%	0.00%
v6	1.3155	23.66%	100.00%	100.00%	0.00%
v7	7.0243	0.03%	94.97%	17.66%	100.00%
v8	23.8581	0.00%	8.91%	44.98%	100.00%
v9	189.9507	0.20%	40.57%	37.40%	1.34%

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	cpi
v1	18.90%	12.60%	2.001
v2	0.00%	11.10%	0.852
v3	19.60%	13.00%	1.624
v4	0.00%	11.10%	0.67
v5	49.90%	9.10%	25.542
v6	22.50%	15.00%	2.177
v7	45.10%	10.00%	1.513

Figure A.5: Mean values for the gadget variants, in the Broadwell environment.

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate	store-page-walks
v1	0.0166	0.0155	3.6422	1.5161	0.0000	0.0000
v2	0.0215	0.5298	0.3205	0.1801	0.0005	0.0149
v3	0.1970	0.4092	0.0000	0.0000	0.4526	3.1002
v4	0.0023	0.0000	0.0033	5.5111	0.0000	0.0000
v5	0.0089	0.0026	0.0060	0.4266	1.0525	1.8783
v6	0.0169	0.0001	0.0000	0.0012	2.8557	3.2511
v7	0.0033	0.0996	0.0008	0.0015	1.5889	0.6787
v8	0.0827	2.8506	0.7560	1.7895	0.0102	0.0160
v9	0.0298	0.7322	1.1496	-4.9284	0.0013	0.0000

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	br-non-conditionals-miss-rate	cpi
v1	0.0023	0.0023	4.1952	0.0072
v2	0.1727	1.1062	1.1927	0.0060
v3	0.0040	0.0037	3.5389	0.0064
v4	0.3380	8.5301	9.4733	0.1522
v5	0.0009	0.0008	1.4770	0.0038
v6	0.0063	0.0064	1.5414	0.0049
v7	0.0034	0.0032	2.1028	0.0160

Figure A.6: CoV values for the gadget variants, in the Kabylake environment.

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate	store-page-walks
v1	0.0000	0.0000	0.0580	0.4529	0.0000	0.0000
v2	0.0011	0.0259	0.0022	0.0060	0.0030	2.6983
v3	0.2677	0.5111	0.0000	0.0001	3.2225	3.0000
v4	0.0003	0.0000	0.0000	1.7399	0.0000	1.0213
v5	0.0005	0.0009	0.0000	0.3901	0.0000	4.3589
v6	0.0031	0.0001	0.0000	0.0004	0.0000	4.3589
v7	0.0011	0.0015	0.0000	0.0003	0.0000	0.2563
v8	0.0002	0.0435	0.0103	0.3187	0.0001	0.7839
v9	0.0181	0.2009	0.0600	0.1088	0.0000	2.6323

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	br-non-conditionals-miss-rate	cpi
v1	0.0023	0.0023	0.8951	0.0011
v2	0.2197	0.0004	0.0008	0.0043
v3	0.0033	0.0025	1.2479	0.0044
v4	0.3048	0.0004	0.0005	0.0006
v5	0.0007	0.0005	2.5913	0.0000
v6	0.0241	0.0240	2.5034	0.0148
v7	0.0011	0.0010	2.0040	0.0006

Figure A.7: CoV values for the gadget variants, in the Haswell environment.

Cache Gadget variant	cpi	l1-load-miss-rate	l2-load-miss-rate	l3-load-miss-rate	l3-store-miss-rate	store-page-walks
v1	0.0003	0.0000	1.0006	0.0000	0.0000	0.0000
v2	0.0042	0.1130	0.2485	0.3157	0.0358	0.7596
v3	0.0401	0.4987	0.0001	0.0000	2.5632	2.9017
v4	0.0035	0.0000	0.0028	2.2361	0.0000	0.0000
v5	0.0011	0.0024	0.0207	5.3852	0.0000	0.0000
v6	0.0068	0.0000	0.0000	0.0000	0.0000	2.2957
v7	0.0035	0.0015	0.0000	0.0000	0.0000	0.6659
v8	0.0003	0.1608	0.0164	0.5455	0.0001	2.1199
v9	0.0310	0.8548	2.4894	0.0000	0.0001	5.3852

BP Gadget variant	br-conditionals-miss-rate	br-miss-rate	br-non-conditionals-miss-rate	cpi
v1	0.038647	0.039541	1.985744	0.018455
v2	0.140843	0.000751	0.001449	0.00242
v3	0.017279	0.018463	2.255612	0.006859
v4	0.23778	0.002496	0.005105	0.003566
v5	0.000898	0.000731	1.700267	0.000551
v6	0.005642	0.005599	1.176661	0.003501
v7	0.001478	0.001483	0.864147	0.000822

Figure A.8: CoV values for the gadget variants, in the Broadwell environment.

BIBLIOGRAPHY

[1] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 307–319.

[2] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the relation of external and internal feature interactions: A case study." In: *arXiv preprint arXiv:1712.07440* (2017).

[3] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-influence models for highly configurable systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 284–294.

[4] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. "Exploring feature interactions in the wild: the new feature-interaction challenge." In: *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. 2013, pp. 1–8.

[5] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-based sampling of software configuration spaces." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1084–1094.

[6] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "Configcrusher: Towards white-box performance analysis for configurable systems." In: *Automated Software Engineering* 27.3 (2020), pp. 265–300.

[7] Xue Han, Tingting Yu, and Michael Pradel. "Confprof: White-box performance profiling of configuration options." In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2021, pp. 1–8.

[8] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. "Causality in configurable software systems." In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 325–337.

[9] Ajay M Joshi, Lieven Eeckhout, Lizy K John, and Ciji Isen. "Automated microprocessor stressmark generation." In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE. 2008, pp. 229–239.

[10] Larry W McVoy, Carl Staelin, et al. "lmbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.

[11] Samuel Kounev, Klaus-Dieter Lange, and Joakim von Kistowski. *Systems benchmarking*. 1st ed. Cham, Switzerland: Springer Nature, Aug. 2020.

[12] Alexey Kopytov. "Sysbench manual." In: *MySQL AB* (2012), pp. 2–3.

[13] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. "A taxonomy of variability realization techniques." In: *Software: Practice and experience* 35.8 (2005), pp. 705–754.

[14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. "Variability in software systems—a systematic literature review." In: *IEEE Transactions on Software Engineering* 40.3 (2013), pp. 282–306.

[15] Sven Apel and Christian Kästner. "An overview of feature-oriented software development." In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[16] Don Batory. "Feature models, grammars, and propositional formulas." In: *Software Product Lines: 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005. Proceedings 9*. Springer. 2005, pp. 7–20.

[17] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering*. Vol. 10. Springer, 2005.

[18] Andreas Metzger and Klaus Pohl. "Software product line engineering and variability management: achievements and challenges." In: *Future of software engineering proceedings* (2014), pp. 70–84.

[19] Ulrich Drepper. "What every programmer should know about memory." In: *Red Hat, Inc* 11.2007 (2007), p. 2007.

[20] Part Guide. "Intel® 64 and ia-32 architectures software developer's manual." In: *Volume 3B: System programming Guide, Part* 2.11 (2011).

[21] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. "A survey of different approaches for overcoming the processor-memory bottleneck." In: *International Journal of Computer Science and Information Technology* 9.2 (2017), pp. 151–163.

[22] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. "Inside 6th-generation intel core: New microarchitecture code-named skylake." In: *IEEE Micro* 37.2 (2017), pp. 52–62.

[23] Mark S Papamarcos and Janak H Patel. "A low-overhead coherence solution for multiprocessors with private cache memories." In: *Proceedings of the 11th annual international symposium on Computer architecture*. 1984, pp. 348–354.

[24] Peter J Denning. "Virtual memory." In: *ACM Computing Surveys (CSUR)* 2.3 (1970), pp. 153–189.

[25] Abhishek Bhattacharjee and Margaret Martonosi. "Inter-core cooperative TLB for chip multiprocessors." In: *ACM Sigplan Notices* 45.3 (2010), pp. 359–370.

[26] Doug Joseph and Dirk Grunwald. "Prefetching using markov predictors." In: *Proceedings of the 24th annual international symposium on Computer architecture*. 1997, pp. 252–263.

[27] T-F Chen and J-L Baer. "A performance study of software and hardware data prefetching schemes." In: *ACM SIGARCH Computer Architecture News* 22.2 (1994), pp. 223–232.

[28] Brendan Gregg. *Systems Performance*. 2nd ed. Addison-Wesley Professional Computing Series. Upper Saddle River, NJ: Pearson, Jan. 2021.

[29] Mark D Hill and Alan Jay Smith. "Evaluating associativity in CPU caches." In: *IEEE Transactions on Computers* 38.12 (1989), pp. 1612–1630.

[30] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[31] James E Smith and Gurindar S Sohi. "The microarchitecture of superscalar processors." In: *Proceedings of the IEEE* 83.12 (1995), pp. 1609–1624.

[32] James E Smith. "A study of branch prediction strategies." In: *25 years of the international symposia on Computer architecture (selected papers)*. 1998, pp. 202–215.

[33] Agner Fog. "The microarchitecture of Intel, AMD and VIA CPUs." In: *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering* (2011).

[34] André Seznec and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction." In: *The Journal of Instruction-Level Parallelism* 8 (2006), p. 23.

[35] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, and Gunter Saake. "Measuring non-functional properties in software product line for product derivation." In: *2008 15th Asia-Pacific Software Engineering Conference*. IEEE. 2008, pp. 187–194.

[36] Mahrukh Umar and Naeem Ahmed Khan. "Analyzing non-functional requirements (NFRs) for software development." In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE. 2011, pp. 675–678.

[37] David Ameller, Claudia Ayala, Jordi Cabot, and Xavier Franch. "How do software architects consider non-functional requirements: An exploratory study." In: *2012 20th IEEE international requirements engineering conference (RE)*. IEEE. 2012, pp. 41–50.

[38] Reham Fadul. "Quantitative Assessment of Nonfunctional Requirements in Product Families." PhD thesis. 2014.

[39] *Systems and software Quality Requirements and Evaluation (SQuaRE)*. Standard. International Organization for Standardization.

[40] Ajay Joshi, Lieven Eeckhout, and Lizy John. "The return of synthetic benchmarks." In: *2008 SPEC Benchmark Workshop*. 2008, pp. 1–11.

[41] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. "A survey on fault injection techniques." In: *Int. Arab J. Inf. Technol.* 1.2 (2004), pp. 171–186.

[42] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. "On fault representativeness of software fault injection." In: *IEEE Transactions on Software Engineering* 39.1 (2012), pp. 80–96.

[43] Norman P Jouppi. "Cache write policies and performance." In: *ACM SIGARCH Computer Architecture News* 21.2 (1993), pp. 191–201.

[44] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. " O'Reilly Media, Inc.", 2005.

[45] André Seznec. "A new case for the tage branch predictor." In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 2011, pp. 117–127.

[46] Thomas J DiCiccio and Bradley Efron. "Bootstrap confidence intervals." In: *Statistical science* 11.3 (1996), pp. 189–228.

[47] Samuel Sanford Shapiro and Martin B Wilk. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611.

[48] William H Kruskal and W Allen Wallis. "Use of ranks in one-criterion variance analysis." In: *Journal of the American statistical Association* 47.260 (1952), pp. 583–621.

[49] Charles G Martin and Paul A Games. "ANOVA tests for homogeneity of variance: Nonnormality and unequal samples." In: *Journal of Educational statistics* 2.3 (1977), pp. 187–206.

[50] T Puzak, Allan Hartstein, Philip G Emma, Vijayalakshmi Srinivasan, and Arthur Nadus. "Analyzing the cost of a cache miss using pipeline spectroscopy." In: *Journal of Instruction-Level Parallelism* 10 (2008), pp. 1–33.

[51] Thorsten Berger and Jianmei Guo. "Towards system analysis with variability model metrics." In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. 2014, pp. 1–8.

[52] Sandro Schulze and Wolfram Fenske. "[Engineering Paper] Analyzing the Evolution of Preprocessor-Based Variability: A Tale of a Thousand and One Scripts." In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2018, pp. 50–55.

[53] Luk Van Ertvelde and Lieven Eeckhout. "Benchmark synthesis for architecture and compiler exploration." In: *IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE. 2010, pp. 1–11.

[54] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. "Tailoring dynamic software product lines." In: *Proceedings of the 10th ACM international conference on Generative programming and component engineering*. 2011, pp. 3–12.