Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Semi-static Conditions in Low-latency C++ for High Frequency Trading: Better than Branch Prediction Hints

Author:
Maximilian Lucuta

Supervisor:
Dr Paul A. Bilokon

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

September 2023

Abstract

Conditional branches pose a challenge for code optimisation, particularly in low latency settings. For better performance, processors leverage dedicated hardware to predict the outcome of a branch and execute the following instructions speculatively, a powerful optimisation. Modern branch predictors employ sophisticated algorithms and heuristics that utilise historical data and patterns to make predictions, and often, are extremely effective at doing so. Consequently, programmers may inadvertently underestimate the cost of misprediction when benchmarking code with synthetic data that is either too short or too predictable. While eliminating branches may not always be feasible, C++20 introduced the [[likely]] and [[unlikely]] attributes that enable the compiler to perform spot optimisations on assembly code associated with likely execution paths. Can we do better than this?

This report presents the development of a novel language construct, referred to as a semi-static condition, which enables programmers to dynamically modify the direction of a branch at run-time by altering the assembly code within the underlying executable. Subsequently, the report explores scenarios where the use of semi-static conditions outperforms traditional conditional branching, highlighting their potential applications in real-time machine learning and high-frequency trading. Throughout the development process, key considerations of performance, portability, syntax, and security were taken into account. The resulting construct is open source and can be accessed at `https://github.com/maxlucuta/semi-static-conditions`.

Acknowledgments

Thank you to Erez Shermer, Founder, CTO & MM at qSpark for proposing and formulating the project. A special appreciation is extended to Dr. Paul Bilokon and the qSpark engineering team for their valuable guidance and consistent feedback during the course of the project.

Contents

1 Introduction **1**
1.1 Aims and Approach . . . 1
1.2 Research Context . . . 1
1.3 Report Outline . . . 2
1.4 Summary of Achievements . . . 3

2 Background **4**
2.1 Pipelining and Conditional Branches . . . 4
2.2 Dynamic Branch Prediction . . . 7
2.3 C++ and Compiler Hints . . . 10
2.4 High Frequency Trading . . . 13

3 Research and Development **15**
3.1 Outline . . . 15
3.2 Semi-static Conditions . . . 15
3.3 Prototype Development . . . 17
3.4 Optimisations . . . 26
3.5 Generalisations . . . 30

4 Benchmarks and Applications **33**
4.1 Outline . . . 33
4.2 Experimental Method . . . 33
4.3 Benchmarks . . . 36
4.4 Applications . . . 45
4.5 Summary of Conclusions . . . 55

5 Evaluation **57**
5.1 Overall Approach . . . 57
5.2 Safety . . . 59
5.3 Reliability . . . 62
5.4 Usage and Portability . . . 63
5.5 Experimental Method . . . 65

6 Ethics **67**

7 Conclusions **68**

Chapter 1

Introduction

1.1 Aims and Approach

The implementation of semi-static conditions, a language construct that can programmatically alter the direction of a branch at execution time, and the identification of instances where it outperforms conditional branching constitute the primary objectives of this project. To achieve this goal, there should be no runtime checks associated with branch-taking. A key challenge in the development process is to create a language construct that mimics the behavior of direct method invocations while simultaneously providing the ability for dynamic switching of branch directions, to facilitate the desired behaviour.

The research contribution of this paper is structured into two stages. In the first stage, the focus is on the development of the construct, with an emphasis on the strategies employed to facilitate low-latency branch-taking through binary editing, addressing the associated considerations of syntax, performance, and portability. The second stage shifts the focus to exploring instances where semi-static conditions out-perform conditional statements, whilst also investigating the implications of runtime assembly modification on hardware behaviour. Additionally, a comprehensive software archive showcasing examples of usage will be created and made readily accessible.

1.2 Research Context

Current research in branch prediction optimization is predominantly concentrated on hardware-based solutions that aim to enhance speculative efficiencies and overall performance of modern CPU's. However, despite these efforts, the problem of branch prediction remains unresolved, with limited attention given to software-based optimizations [1]. For typical commercial applications, the pursuit of such micro-optimizations is often unnecessary and may introduce additional complexity. Nonetheless, in industries like High Frequency Trading (HFT), even slight improvements in execution latencies on the clock cycle level are highly valued. Consequently, these optimizations are highly sought-after and can provide significant competitive advantages. Due to their crucial role in determining a firm's profitability and success,

cutting-edge research on software-based optimizations in such industries is typically shrouded in secrecy.

Several books have attempted to bridge the divide between computational and financial research, aiming to combine mathematical modeling and the development of algorithmic trading strategies (e.g., [2, 3]). Additionally, numerous public conferences are available, focusing on the development of low-latency execution systems (e.g., [4, 5, 6, 7]), emphasizing topics such as data structures, atomics, and low-latency design patterns. Notably, there has been some emphasis on branchless design, which showcases some common alternatives to conditional statements with high misprediction rates [8]. However, the strategies employed in this context lack flexibility and rely on the assumption that branches can be pre-computed without incurring significant costs.

In contrast, there is a wealth of literature and extensively documented resources available for C++ [9, 10], which is widely used as the primary language in the development of low-latency trading systems. However, when it comes to the development of semi-static conditions and strategies involving the modification of running executables using C++, the scope becomes more specialized. Nevertheless, these techniques are well-documented and find applications in various areas such as debuggers, profilers, hot patching software, and security tools (e.g., [11, 12, 13]).

There is a significant scarcity of ultra-low latency C++ tools, particularly those specifically designed to address control flow problems, offering both rigorous application verification and superior performance. While it is possible to come across online posts outlining small-scale experiments that focus on minor branch optimizations in specific scenarios (as demonstrated in [8]), such micro-optimizations are often overlooked and left to the compiler and hardware to handle. Interestingly, extensive research has been conducted on the true cost of branch misprediction [14, 15], highlighting its significant contribution to performance bottlenecks in low-latency systems. While these articles provide in-depth analysis of benchmark data and performance penalties, the strategies proposed for preventing branch mispredictions remain lackluster or non-existent.

In light of existing literature and its insights into software-based branch optimization problems, it becomes apparent that a research gap exists, which this study aims to fill. The motivation behind this research lies in providing solutions to the aforementioned void and contributing to the understanding of software-based branch optimization.

1.3 Report Outline

In Chapter 2, the literature review section of this report, the focus is on encapsulating and critically analyzing research pertaining to the problem at hand. The section begins with an outline of modern CPU pipelined architectures and the implications (and cost) of branched execution. Next, attention is shifted to advancements in hardware-based solutions, outlining the strengths and weaknesses of various schemes when encountering branches of different predictability. The focus is then redirected to software, providing an outline of C++ and its importance in

the development of low-latency trading systems. Language features that exist exclusively to optimize branch prediction are also discussed. Finally, discussions are concluded with HFT, examining economic effects, known technical advancements and the problems that remain to be solved in the industry.

Chapters 3 and 4 are dedicated to the research contribution of this report, which can be conceptualized as consisting of two stages: the development of semi-static conditions and data-backed applications. In the development stage (Chapter 3), a sequential approach is adopted to identify the requirements and challenges associated with designing the language construct. The solution to the problem is outlined and demonstrated, providing an overview of key theory with subsequent design decisions and optimizations. The proceeding stage (Chapter 4) demonstrates the instances where semi-static conditions offer superior performance compared to conditional branching, accompanied by detailed analyses and supporting benchmark data to substantiate the findings, alongside novel investigations into effects of binary editing on modern hardware. Furthermore, discussions are conducted on how the outlined scenarios can be incorporated into a commercial trading system, considering their suitability and practical implications.

In Chapter 5, the software contribution alongside the various experimental methods employed to benchmark semi-static checks are critically evaluated. Detailed examples of usage are provided, along with recommendations for maximizing the security and reliability of the language construct. Chapter 6 provides a brief discussion of the ethical implications of this research, with concluding remarks in Chapter 7, discussing potential areas for further research and development.

1.4 Summary of Achievements

The project has achieved significant milestones across multiple dimensions. The research contribution offers valuable insights and approaches for developing software-based branch optimizations, with comprehensive and rigorous investigations into the resulting hardware level behaviours, filling an important research gap in the academe. By employing an unconventional yet effective binary editing strategy, the project enables ultra-low latency branch execution through the decoupling of condition evaluation logic and branch taking, controlled directly by the programmer. Through extensive benchmarking and detailed performance analysis in pseudo realistic scenarios, the proof-of-concept semi-static checks demonstrate their real-world applicability, particularly within real-time systems and high-frequency trading.

The research contribution is encapsulated within a open-source library that allows programmers to utilize the language construct for both commercial and experimental purposes. With a focus on syntax, security, efficiency, and portability, semi-static conditions can be seamlessly integrated into high-performance real-time systems, as exemplified in domains such as high-frequency trading and real-time machine learning. This report has since gained significant traction in the high-performance computing space, being featured on the premier newsletter for performance tuning, Easyperf, and considered for publishing by the prestigious Journal of Parallel and Distributed Computing [16].

Chapter 2

Background

2.1 Pipelining and Conditional Branches

The microprocessor is an integrated circuit responsible for executing arithmetic, logic, control, and I/O operations in a digital system [17]. The early 1970's saw the emergence of the first commercially available microprocessor, the Intel 4004, initially designed as a 4-bit 740 kHz central processing unit (CPU) for early printing calculators [18, 19]. Over the past three decades, advancements in integrated circuit technology driven by the exponential growth in transistor density have enabled microprocessor manufacturers to develop increasingly sophisticated CPU's. [20]. Alongside these developments, the introduction of modern instruction sets and standardized operating systems have propelled the computational capabilities of contemporary computers to unprecedented heights.

Modern CPU's utilize pipelining as an implementation technique to exploit the inherent parallelism in instruction execution to improve overall throughput [17]. Similar to cars on an assembly line, pipelining allows for the overlapping execution of multiple instructions, with each step in the assembly line constituting a pipe stage that represents a phase in the fetch-decode-execute cycle. If all stages take the same amount of time, a pipeline with n stages will achieve a throughput n times greater than an un-pipelined counterpart, with the bottleneck stage bounding the number of processor cycles required for a single execution [17]. Whilst Figure 2.1 provides a simplified schematic of instruction pipelining, it is important to note that modern processors have vastly more complex pipelined architectures, often super-scalar with varying instruction sets and addressing modes to prevent memory-access conflicts in instruction/data memory, optimize register handling, and maximize instruction level-parallelism [21].

Pipelining in CPU's, while a powerful optimization technique, introduces hazards that can impact overall performance. These hazards can be broadly categorized into three types: structural hazards, data hazards, and control hazards [17]. (1) Structural hazards occur when hardware resources are incompatible with the sequence of instructions leading to conflicts in resource allocation, resulting in pipeline stalls. However, advancements in super-scalar technology and out-of-order instruction execution have made structural hazards less prevalent to virtually non-existent [17].

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction $i + 1$	IF	ID	EX	MEM	WB				
Instruction $i + 2$		IF	ID	EX	MEM	WB			
Instruction $i + 3$			IF	ID	EX	MEM	WB		
Instruction $i + 4$				IF	ID	EX	MEM	WB	
Instruction $i + 5$					IF	ID	EX	MEM	WB

Figure 2.1: Simplified representation of 5 stage pipeline using a RISC instruction set. On each clock cycle, a new instruction is fetched and all proceeding instructions progress to a new pipe-stage in the fetch-decode-execute cycle, achieving a throughput fives times greater than a non-pipelined counterpart. Instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM) and write back (WB). Table has been taken and adapted from [17].

(2) Data hazards arise due to dependencies between instructions that have not finished executing, for example, relying on register data that preceding instructions manipulate. These dependencies can cause conflicts and hinder parallel execution, but are broadly mitigated through the use of virtual registers (register renaming) [21]. (3) Control hazards are caused by branch instructions that change the program counter. These instructions introduce uncertainty into the execution flow since the branch target needs to be determined prior to the next instruction fetch [17]. Branches constitute around 12-30% of all instructions executed on modern instruction sets and are widely regarded as the most significant barrier to achieving single cycle executions [22], and as a consequence, have become a large area of focus for optimisations by hardware and software engineers.

Microprocessors employ various strategies to mitigate control hazards in the CPU instruction pipeline. The simplest and most costly approach is a full pipeline stall/freeze, where proceeding instructions after a change in PC are ignored until the target of the branch is known, resulting in a fixed cycle penalty [17, 23]. Improving upon this, processors can make static predictions about the branch target instead of discarding subsequent instructions, maintaining sequential execution of instructions pertaining to either the *taken* or *not-taken* branches. By leveraging compiler static analysis and optimizing likely paths of execution, static prediction becomes a powerful optimization technique in pipelined processors, providing non-zero probabilities of correctly predicted branch targets and thus minimising throughput loss from flushes [22, 24]. Though an improvement, this approach is rather inflexible particularly for branch targets that change. Static prediction schemes lack the ability to adapt at runtime to changing patterns in branch target execution which is a common theme for the majority of conditional branches. With modern CPU's employing increasingly more speculative architectures and deeper pipelines to maximise instruction throughput [17], branch penalties become more significant and scale monotonically with pipeline depth [25]. With this in account, it is clear that pro-

Untaken branch instruction	IF	ID	EX	MEM	WB			
Instruction i + 1		IF	ID	EX	MEM	WB		
Instruction i + 2			IF	ID	EX	MEM	WB	
Instruction i + 3				IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB			
Instruction i + 1		IF	**idle**	**idle**	**idle**	**idle**		
Branch target			IF	ID	EX	MEM	WB	
Branch target + 1				IF	ID	EX	MEM	WB

Figure 2.2: Example of a five stage RISC pipeline when a branch instruction is correctly predicted to be *not-taken* (top), resulting in subsequent instructions to fall through without hindering overall throughput. Branch target mispredictions (bottom) cause the pipeline to be flushed resulting in idle cycles. Table has been taken and adapted from [17].

cessors require even more aggressive optimisation techniques beyond simple static analysis to minimise idle cycles from branch mispredictions.

The final mitigation technique utilised commonly on classical 5-stage MIPS architectures, but are typically non-existent on modern processors, are *branch delay slots* which interleave instructions independent to the branch prior to branch target deduction at decode or execute time [26]. The concept is rooted in the observation that not all instructions in a program depend on the outcome of a branch instruction, thereby in theory, allowing the compiler to schedule instructions before the branch is taken and hide some of the branches latency. However, on modern processors, branch delay slots are generally avoided. Utilizing branch delay slots effectively requires the compiler to identify and schedule instructions that can fill the slots, adding complexity to the compiler design and optimization process. Compiler writers need to analyze dependencies, identify independent instructions, and rearrange code to take advantage of the delay slots. This additional burden makes it more challenging to generate efficient assembly and can increase compilation time [22, 26]. With deeply intricate instruction pipelines on modern processors, the scheduling task becomes exponentially more complex and interferes with modern hardware solutions, deeming the once performance-enhancing technique as a complication in modern processors.

Whilst this section offers a fundamental overview of conditional branches and their implications at the processor level, it is crucial to recognize the significance of primitive branch penalty mitigation techniques. These early forms of mitigation laid the groundwork for modern solutions discussed in subsequent sections, with the evolution of speculative processors playing a pivotal role in driving the development of powerful hardware-based branch predictors.

2.2 Dynamic Branch Prediction

The advancements of super-scalar technology have introduced increasingly speculative architectures with deep instruction pipelines, effectively maximising throughput to meet the performance requirements of contemporary computers. Consequently, the issue of branch mispredictions have remained a significant impediment to sequential instruction execution [17, 25]. As briefly mentioned earlier, static branch prediction techniques rely on predetermined rules or assumptions rather than leveraging runtime information. To enhance static prediction on modern architectures, compilers play a vital role in making profile-guided decisions based on historical execution patterns. Notably, the binomial distribution of simple branches renders static prediction an effective strategy [17, 27]. Supporting this notion, Fisher and Freudenberger's work demonstrates that applications with statically predictable branches exhibit commendable performance under the current paradigm [28]. The overarching limitation of the former schemes is the inability to adapt to runtime changes, varying input conditions, or capture complex patterns in branch behavior. Whilst previous execution patterns can be used as a proxy to determine the likelihood of a branch, in a real time system with non-deterministic behaviour, solely relying on such a scheme would likely lead to higher mispredictions rates and performance degradation. In light of these issues, a plethora of dynamic branch predictors where developed with the ability to adapt, leverage runtime information, and capture complex branch patterns to improve prediction accuracy and overall performance.

Dynamic branch predictors (BP) can be broadly categorized into one-level *local* BP's and two-level *global* BP's, with more modern BP schemes incorporating the strengths of both. One-level BP's typically utilize a one-dimensional *branch prediction buffer* or *branch history table*, acting as a cache indexed by the lower bits of the program counter associated with a branch instruction [17, 29]. 1-bit prediction schemes have a single bit entry in the branch history table, indicating whether the branch has recently been taken or not. In the event of a misprediction, the prediction bit is inverted. While this scheme offers simplicity and low hardware overhead, the limited historical information stored in a single bit often leads to frequent mispredictions [17]. To address this limitation, *N-bit* saturated counter schemes are statically assigned to branches with distinct addresses. When a branch is about to be taken, the counter value associated with the branch is used to select the branch target based on a predetermined threshold [27]. The count is incremented if the branch is taken, and vice versa. It may seem intuitive that increasing the number of bits would improve prediction accuracy through capturing more historical information, however Smith demonstrated that counter sizes larger than 2-bit schemes do not consistently result in higher prediction accuracies [30]. For loop insensitive programs with biased branches, 2-bit counter schemes have been found to be effective with branch misprediction rates averaging at 11%, but performance was found to deteriorate with integer based programs with more complex branch dependencies [17]. The limitations of one-level BP schemes are multifaceted. Relying on a single branch history table restricts the ability to capture intricate patterns and dependencies between branches, hampering the prediction accuracy of one-level BP's [30]. Additionally, interference can occur with branch buffer accesses, as finite space ne-

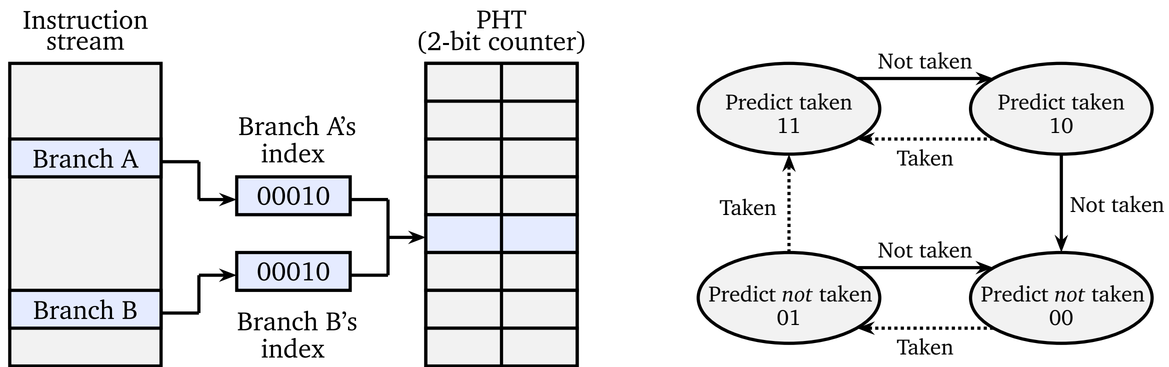


Figure 2.3: Diagrammatic representation of a one-level branch history table with aliasing interference between two branches, taken and adapted from [29] (left). State machine for 2-bit branch prediction, taken and adapted from [17] (right).

cessitates the use of hashing schemes to access the bit-counters for predictions. This can lead to collisions, with negative aliasing occurring more prominently than positive aliasing [29].

In light of these limitations Yeh and Patt proposed the first two-level adaptive branch prediction scheme, utilising a *global branch history table* that maintains a shared history of branch outcomes, allowing it to capture patterns and dependencies between branches [31]. Implementation wise, two level BP's comprise of a branch history register (BHR) which tracks recent outcomes of branches and a global pattern history table (PHT) which stores patterns and outcomes associated with specific branch instructions. In this scheme both the BHR and PHT work in collaborative fashion; most recent branch results are shifted into the BHR with branch addresses being used to index a BHR table, the content of the BHR is used to index the global PHT for making predictions, with mispredictions updating both the PHT and BHR [29, 31, 32]. Whilst the two-level adaptive predictor improves prediction accuracy and captures branch dependencies, it still faces challenges related to aliasing conflicts similar to saturated counter-based branch predictors. Yeh and Patt explored several alternative branch prediction schemes based on the original two-level approach, and whilst branch correlation algorithms could be improved it was found that inherent trade-offs existed between efficient storage capacity, memory access overhead and reduced interference, often bounded by physical constraints [32]. As a consequence, current state-of-the-art BP research is focused primarily on the development and optimisation of the prediction algorithms that build off the foundational work originally done by [31, 32], and have been immensely successful at doing so with the emergence of incredibly powerful BP's such as the competition winning TAGE-L with 3-4 mispredictions per 1000 instructions [33, 34].

Whilst the current state of modern BP's are capable of predicting the majority of branches to near perfect accuracy, there exists classes of branches that are inherently hard to predict (HTP) even by TAGE-like predictors, and hence the problem of branch prediction is still considered unsolved [1, 37]. The key weakness arise from attempting to identify correlations between branches in a noisy global history, or

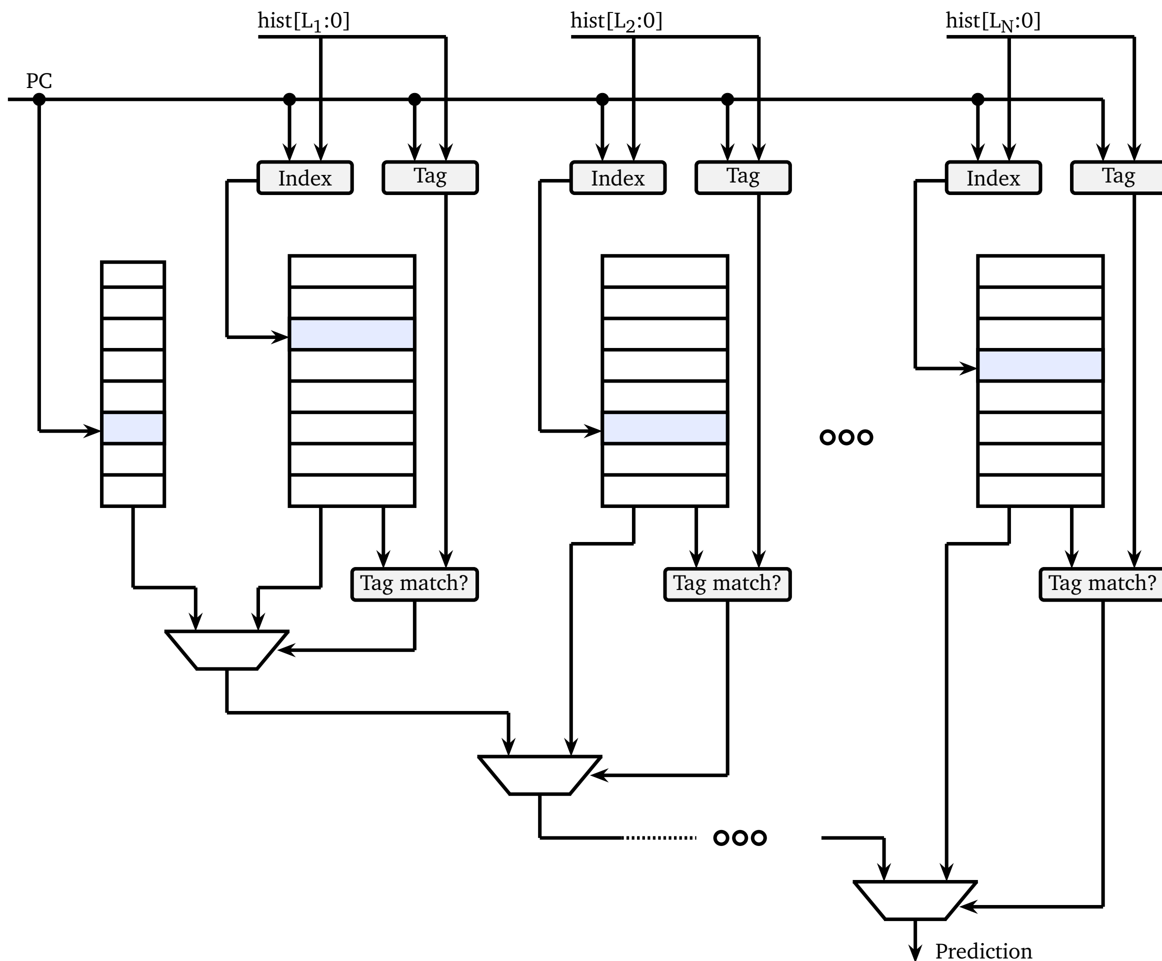


Figure 2.4: Organisation of the TAGE (Tagged Geometric) BP with n-tagged tables. TAGE features a base 2-bit binomial predictor that provides a basic prediction, and a number of partially tagged tables that store branch history information. At prediction time, each of the tagged tables are indexed simultaneously using global and local branch histories, with the longest tag match generally being chosen to make a prediction [35]. The TAGE-L predictor builds on this and uses a dynamic table organization with varying numbers of tables and table lengths to better capture branch outcomes and histories [36]. TAGE-SC-L further builds on this and incorporates a statistical correlator to further refine predictions [33]. Figure taken and adapted from [35].

the branches themselves are uncorrelated and must rely solely on saturated counter schemes to make predictions [34]. Fundamentally, noise caused by HTP branches introduce a number of redundant patterns which pollute the global branch history causing TAGE-like predictors to struggle, and with non-deterministic orderings of historical patterns, more exotic perceptron based BP's which rely on the position of a branch in the global history also struggle [29, 34, 38]. Problematically, HTP branches are common in most applications but have the more prevalent effects in performance-critical fields such as games, streaming services, and HFT.

The weaknesses of modern BP's stem from their need to be simple, computation-

ally cheap and adaptive to execution phase behaviour [34]. The worst case storage requirements for capturing branch patterns is $\mathcal{O}(2^n)$, posing a physical constraint on the predictive capabilities of modern BP's. Whilst fundamental breakthroughs that build off TAGE and perceptron-based BP's have become rare, Zangeneh et al. proposed a convolutional neural network (CNN) BP which can be trained offline [34]. Their work explored two CNN models, one of which was a pure software solution and the second being a smaller hardware optimised version capable of being implemented on chip, both of which showed vast reductions in MPKI on benchmarks run on TAGE-SC-L [34]. Although prediction accuracy greatly improved for correlated branches with noisy histories, BranchNet showed poor performance on data-dependant branches and programs where mispredictions are spread across a number of static branches. The reason for the former is that data dependant branches rely on input data and program phase behaviour, meaning there is little to no branch history that can be correlated to data stored in memory, a problem for capturing training data to make predictions [34]. In terms of mispredictions for static branches, this is a problem regarding the sparse storage requirements for a practical CNN BP, though it may be possible to implement larger models using proposed predictor virtualisation techniques [39].

CNN based BP's have the potential to be the next standard for hardware-based BP's, however there is still much work to be done before commercialisation. Whilst the work done by Zangeneh et al. demonstrated MPKI reductions on several HTP branches across a number of benchmarks, the first challenge of using such CNN's would be to achieve generalisation for commercial purposes. This would require a substantially large model (or number of models), introducing additional administrative overhead to the OS [34]. In addition, modern OS's would need to be adapted to load these CNN models on-chip prior to execution time and handle context switching (and its associated penalties) accordingly, in turn adding more complexity and overhead.

At its current state research pertaining to novel hardware based BP's have slowed substantially, and it is likely that modern BP's have reached an asymptotic state of prediction accuracy versus complexity and implementation overhead. Whilst there may be BP's in the future that can have low misprediction rates on noisy or inherently HTP branches, there will always exist some class of branches, especially in real-time systems, that will always be completely non-deterministic and impossible to predict through conventional methods. Whilst this likely can never be solved by hardware, it may be possible that programmatic approaches exist that are able to tame these impossible-to-predict branches.

2.3 C++ and Compiler Hints

High-frequency trading (HFT) demands ultra-low latency and exceptional performance, necessitating the selection of a programming language that offers efficient execution, deterministic behavior, and fine-grained control over hardware resources. C++ has emerged as the primary language for developing critical path (path of order action) components in automated trading systems, primarily due to its design

philosophy. One fundamental axiom associated with C++ is the ”zero overhead principle” [9, 10], which ensures that developers only pay for what they use, resulting in a predictable and transparent performance model. For example, unlike high-level languages that employ garbage collectors to manage memory, C++ utilizes manual memory management. This approach avoids the unpredictable invocation times and overhead associated with garbage collectors, which is critical in low-latency applications that require deterministic performance. C++ provides developers with granular control over memory through abstractions such as pointers, references, heap allocation operators, and standard library methods. However, the trade-off for low-level memory access is increased complexity and the potential for memory leaks and errors. Whilst uncommon, other languages that target the java virtual machine (JVM), for example Java, have become more popular in the HFT space using optimised garbage collection algorithms and compilation techniques to minimise latency cost [40]. Notably, this has become popular with leading quantitative trading firm and liquidity provider, Jane Street.

C++ features also support the shifting of execution time operations to compile time, resulting in the deferral of computational costs and the reduction of runtime latency when appropriately utilized. This capability proves to be a powerful tool for applications with stringent requirements for low latency. Templates, as a Turing-complete language feature, play a vital role in enabling this paradigm by providing type-safe parameterized blueprints, which allow the generation of specialized code by the compiler for each type-specific instantiation. Consequently, compile-time polymorphism can be achieved, albeit at the expense of flexibility and maintainability [41, 42]. Moreover, the template system can be further leveraged for performing recursive instantiations and type deductions, thereby facilitating the manipulation of types and values and enabling the execution of complex computations at compile time. It is important to note that this approach, known as ”template meta-programming,” is extensively employed in low-latency settings such as High-Frequency Trading [5, 43].

In the realm of optimizing branch prediction at the language level, specific extensions provided by compilers have long existed, allowing programmers to provide hints for branch prediction. These hints enable targeted optimizations on anticipated execution paths. In GCC and Clang, this capability is manifested in the form of the `__builtin_expect` attributes, which allow programmers to specify conditions and associated probabilities (fixed as either 0 or 1) for condition evaluation at runtime [44, 45]. It is worth noting that these so-called branch prediction hints do not directly affect the hardware-based branch predictors and have been ineffective at doing so since the release of Intel's Pentium M and Core 2 processors [46]. Instead, compiler built-ins optimize branch-taking by rearranging assembly code related to branches to exploit processor static prediction schemes and instruction cache effects for improved performance.

Modern processors enhance execution performance by prefetching instructions sequentially from slower to faster memory storage, such as high-speed caches in close proximity to the CPU. This prefetching is done to avoid high memory access latencies during the fetch stages [47]. When executing a code segment containing

```
if (condition)                    10af:  je     10bd
    function_1();                 10b1:  call   <function_1>
else                              (...)
    function_2();                 10bd:  call   <function_2>
```

Figure 2.5: Comparison of C++ code without branch prediction hints (compiled with GCC on x86-64). In this case the forward branch (else) is assumed not taken and the backward branch (if) is assumed taken.

```
if (condition) [[unlikely]]       10af:  je     10bd
    function_1();                 10b1:  call   <function_2>
else                              (...)
    function_2();                 10bd:  call   <function_1>
```

Figure 2.6: Comparison of C++ code with branch prediction hints (compiled with GCC on x86-64). Since the backward branch is now deemed as ”unlikely”, the compiler will reorganise the ASM such that the backward branch is now the forward branch and vice versa, in contrast to Figure 2.5.

conditional statements, blocks of sequential assembly instructions associated with different branches are prefetched into the instruction cache, irrespective of how frequently the branches are executed. However, prefetched code that remains infrequently accessed throughout the program’s lifespan can contaminate the instruction cache and result in cache-line fragmentation of hot code segments that are frequently executed. This can introduce jitter and latency costs [48, 49]. To optimize branch prediction, the compiler reorganizes the underlying assembly code such that the conditional jump occurs on the least likely path. This is because modern processors initially assume that forward branches are never taken and, therefore, avoid misprediction by fetching the likely branch to the branch target [46]. It is important to note that such static-prediction schemes are employed when the BPU encounters a branch that has not been previously visited. The dynamic prediction schemes outlined in the previous sections will begin to dictate which blocks are speculatively fetched once a branch pattern history has been established.

C++20 introduced the `[[likely]]` and `[[unlikely]]` attributes, which serve as wrappers around the original `__builtin_expect` attributes and function in the same manner as depicted in Figure 2.6. Apart from this, there have been no other language features attempting to optimize branch prediction [50]. There is limited formal research investigating the performance impact using different benchmarks. However, some online blogs report performance gains (e.g., [48] reports a 15% increase) primarily on branches specifically designed to be highly predictable for demonstration purposes rather than from a research perspective. The fundamental problem with these attributes is that they rely on the programmer’s predictive ability of likely execution paths. While profiling and synthetic benchmark data may offer insights into hot/cold branches, programmers tend to significantly underestimate the

cost of misprediction when misusing these attributes (for example, a misprediction is likely followed by a cache miss due to the infrequent execution of the unlikely path) [15]. Furthermore, the capabilities of these attributes are confined to compile time, rendering them inflexible to change during program execution. If a branch is indeed more likely to be taken at runtime, the programmer will reap the benefit of the discussed prediction and prefetching semantics. However, if the likelihood of the branch changes during execution, the programmer would have no control over it and would suffer from increased latency through assembly reordering schematics. Real-time systems, which constantly need to react to live events and data, inherently exhibit such variability. Consequently, static language features like these are inadequate as effective branch misprediction mitigators. This inadequacy forms the focus of the research conducted in this report.

2.4 High Frequency Trading

The evolution of computer-based trading dates back several decades, starting with the introduction of fully electronic trading by NASDAQ [2]. With the decrease in regulation and advancements in electronic exchanges and telecommunications infrastructure, high-frequency trading (HFT) has gained significant popularity, accounting for over 50% of trading volume in equity markets [51]. Defining HFT itself poses challenges. Haldane [52] emphasizes the use of sophisticated algorithms as its main characteristic, while MacKenzie [53] and Arnoldi [54] highlight the importance of speed in data processing and execution rather than the underlying strategies employed. This report primarily focuses on optimizations that target proprietary automated trading systems, and hence considers HFT as a form of algorithmic trading who's strategies rely on low order execution latencies to ensure profitability.

High-Frequency Trading (HFT) firms actively engage in generating trading signals, validating models, and executing trades in order to exploit inefficiencies in market micro-structure within short time frames, with the ultimate goal of achieving profitability [2]. These firms employ diverse strategies to generate profits in financial markets. One prevalent strategy is market-making, where HFT participants continuously provide liquidity by simultaneously placing buy and sell orders, aiming to profit from the bid-ask spread. Another strategy, known as statistical arbitrage, involves capitalizing on transient deviations from fair value by identifying mispricing opportunities. Additionally, event-driven trading strategies focus on leveraging informational asymmetries that arise from significant market events [55]. While a significant body of literature exists on the economic effects of HFT (e.g., [56, 57, 58]), information pertaining to the engineering aspects of low-latency automated trading systems is often concealed due to reasons discussed in Chapter 1. Nevertheless, some online conferences vaguely present important features of HFT systems, such as the networking stack, kernel bypass, custom hardware, and strategies for enhancing the speed and efficiency of production code [4, 5]. An example of a typical HFT system can be seen in Figure 2.7, outlining the relevant components and order flow.

While this report primarily addresses software-based optimizations for latency-critical applications, it is crucial to underscore the significance of the hardware stack

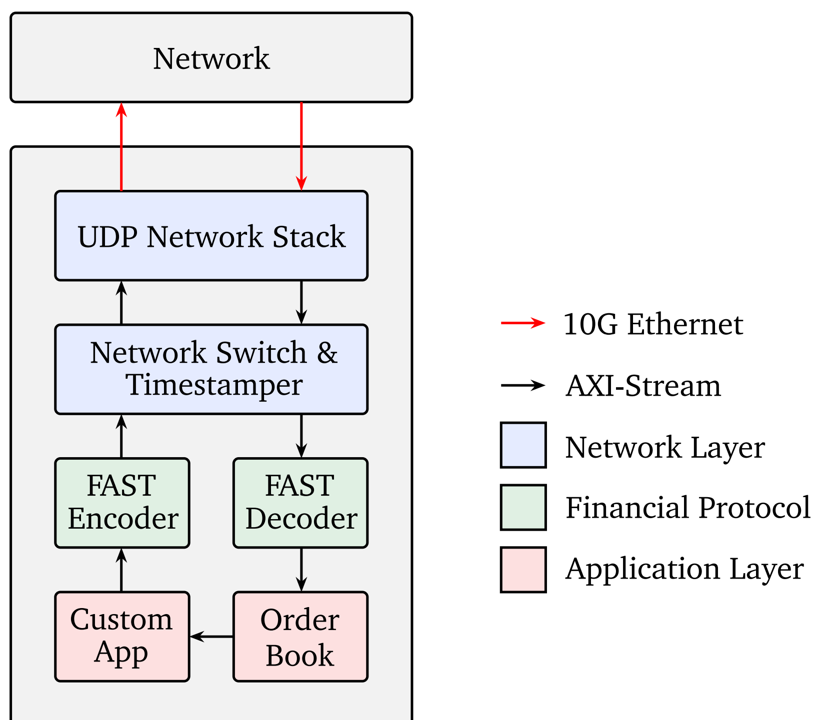


Figure 2.7: Simplified anatomy of a HFT system. Exchanges broadcast ticker data along a 10 GiB Ethernet cable to the HFT system, where the networking stack receives and processes packets in user space. Packets are typically compressed in a domain specific format for bandwidth reasons, which are then parsed into meaningful market orders by the financial protocol which are then ordered in the order book. The custom application then issues the buy and sell orders back over the network, this is the area in which the optimisations presented in this report pertain to. Figure taken and adapted from [59].

in HFT firms for trade execution. Simple trading strategies often rely heavily on the networking stack, implemented entirely on custom firmware and Field Programmable Gate Arrays (FPGA) to achieve execution latencies in the nanosecond range [60]. As proprietary software approaches a state of complete optimisation, it is likely that the engineering focus will increasingly shift towards the hardware stack, which is still in its early stages and has considerable room for growth.

Chapter 3

Research and Development

3.1 Outline

This chapter is dedicated to the idea and development of semi-static conditions, outlining theory, design philosophy and optimisations that have been applied iteratively over the development process. The majority of this chapter is focused on developing a prototype that emulates the behaviour of a simple `if/else` statements, with some discussions given on generalisations to switch statements and non-static member functions in later sections. Whilst discussions on portability are reserved for the evaluation stage, there was an important decision to be made about the choice of development environment for the prototype, more specifically the choice of operating system, compiler and C++ version. Since the primary applications of this language construct will find use in low-latency environments such as HFT systems, the choice of development environment was curated to reflect an industry standard. In light of this, development was done under a Linux OS (Ubuntu distribution) with GCC 13.1 and C++20. At certain stages of the development process, attention will be brought on specific Linux system calls that involve manipulating page permissions for running executables, however equivalent API's exist in other OS's that use paging for virtual memory management.

3.2 Semi-static Conditions

Semi-static conditions can be defined as a language construct that emulate the behaviour of conditional statements, but separates condition evaluation logic and branch taking (the subsequent machine code executed on the pretext of the condition). At compile time, the underlying assembly of semi-static branch taking would resemble that of a function call with no indirection, allowing for a deterministic flow of execution with full support of compiler optimisations and hardware intrinsics. In the context of conditional branching, semi-static conditions **remove the runtime check** associated with branched execution, allowing for faster *branch-taking* in low-latency settings. The *semi* part is associated with the polymorphic nature of the language construct: the ability to programatically change the direction of the branch at runtime, whilst maintaining the deterministic compile-time behaviour already men-

tioned. With semi-static conditions, the lines between the compilation and execution phases of a program become blurred, and the nature of the executable shifts from static to somewhat polymorphic or self-modifying. This behaviour manifests itself within the branch-taking mechanism as a single/sequence of assembly instructions that redirect control flow to the respective `if` or `else` branches, controlled by branch-switching logic that performs the modification.

In the context of branch optimisation, the philosophy behind the construct is simple. Semi-static conditions remove the runtime check from conditional branches and defers this to an auxiliary method which is directly controlled by the programmer, creating an important decoupling of relatively cheap (branch-taking) and expensive (branch-changing) operations. Isolating branch-switching logic in less performance critical code paths allow conditions (more specifically, those pertaining to hard-to-predict branches) in performance-critical sections to be evaluated preemptively without interference, bypassing the need for branch prediction and eliminating misprediction penalties in the latency-critical path. When the hot-path is executed, no conditional checks are needed and the branch is executed as if it where always perfectly predicted. In instances where code paths are infrequently executed, but contain branches that are often mispredicted, semi-static conditions show promise for optimisations. In addition, the overall cheaper branch-taking that semi-static checks offer show promise for more general use cases, even when branches are well predicted.

In order to realise this language construct, some key challenges are addressed in the development process, which can be broadly split into branch-changing and branch-taking logic. Branch-changing logic needs to be able to find the address of the assembly code instructions to edit in memory, and perform the editing in way that calling the branch-taking method redirects program control flow to user-specified regions based on a runtime condition. Branch-taking logic needs to ensure that control flow is redirected with minimal overhead, so it becomes comparable with the execution latency of a perfectly predicted branch, or a direct function call. At a language level, this is not only dependant upon the underlying assembly instructions that are edited, but also being able to reap the full benefits of compiler optimisations without compromising the safety of the program. On the hardware level, it is paramount that branch-taking code benefits from the same caching, instruction pre-fetching and branch target resolution effects that regular function calls or unconditional jumps do to ensure deterministic and low-execution latencies. Whilst it may be possible to achieve near-identical execution schematics to direct function calls on a language level, the cost of cache incoherence and instruction pipeline stalls can trump branch-misprediction penalties by orders of magnitude, making the construct infeasible in low-latency settings. Therefore, it is crucial that semi-static conditions are compatible in this way with modern hardware and processors. Lastly, and more broadly, the language construct needs to be designed with ease of use in mind. This includes simple and elegant syntax, flexibility and a design that allows it to be easily portable across different architectures, compilers and operating systems.

3.3 Prototype Development

The first step in the development process is to establish the desired syntax of the core branch-switching and branch-taking functionality of the language construct. It is likely that this will have a significant influence on the design of semi-static conditions, so establishing how the end product is desired to look in the preliminary stages gives a clear direction in development goals. After careful consideration of simplicity and elegance, the desired usage can be seen below.

```
void function_1() { ... }
void function_2() { ... }
(...)
BranchChanger branch(function_1, function_2);
branch.set_direction(condition);
branch.branch();
```

Semi-static conditions will manifest itself as the `BranchChanger` class which is instantiated by taking the addresses of two functions as arguments. These functions represent the `if` and `else` branches respectively, and their equivalent usage with conditional statements can be seen below.

```
void function_1() { ... }
void function_2() { ... }
(...)
if (condition)
    function_1();
else
    function_2();
```

The `set_direction` method will be responsible for controlling which of the branches is executed based on a user specified runtime condition, whilst the `branch` method will be responsible for executing the branch with minimal overhead. The signature of the `branch` method will always be identical to the functions passed as arguments when the class is instantiated, serving as a single entry and exit point for both branches.

Now that a clear high-level design has been established, the next course of action is to implement the branch-taking functionality. Before delving into assembly instruction modification to facilitate the execution of the `if` and `else` branches, some thought needs to be given in how the `branch` method can act as a single entry and exit point for both branches. This method will not do any meaningful work, its sole purpose is to behave as a trampoline to other areas of the code segment while still being able to propagate return values and register data as if one of the branches where called directly. Its first clear responsibility is to set up the call stack in the exact way that the branches would as if they where called in isolation; since control

flow will be redirected before the `branch` function has opportunity to manipulate the stack, in theory the target branches will be able to use any caller-saved data as if it where called directly. To test this theory, we can observe the disassembly for two functions with identical signatures under **-O0** optimisations so any calling behaviour is not omitted.

```
int add(int a, int b) { ... }
int branch(int a, int b) { ... }
(...)
mov     esi, 2
mov     edi, 1
call    add(int, int)
(...)
mov     esi, 2
mov     edi, 1
call    branch(int, int)
```

As expected, both instances follow a standardised calling convention resulting in identical caller behaviour: arguments are pushed from right to left (in this case since there are less than 3 arguments, they are instead moved into registers as per x86 calling conventions) before the subroutine is executed. While this may seem trivial, the standardisation of calling conventions is an extremely important feature of modern compilers that can be leveraged to generalise the `branch` method in a safe and portable manner.

Now that it's clear that functions with identical signatures observe identical caller setup, and hence the assembly generated on the callee side will be tailored to reflect this, the next course of action is to make the `branch` entry point mimic the identical signature of the branches passed into the constructor. Using class template deduction, return types and arguments of the branches passed into the constructor can be deduced at compile time and leveraged to generate the correct assembly code for the `branch` method.

```
template <typename Ret, typename... Args>
class BranchChanger
{
    using func = Ret(*)(Args...);
    (...)
    BranchChanger(func if_branch, func else_branch);
    (...)
    Ret branch(Args... args);
}
```

The templating splits the signature into arguments and return type, where the arguments are represented by a variadic parameter pack deduced through the pointer

types passed into the constructor (represented with the type alias `func` for readability). These types are then used to declare the signature of the `branch` method, ensuring that it is identical to the `if` and `else` branches. For implementation, virtually anything can be placed inside `branch` and as long as the return type matches the signature, it will compile. Here there are two main cases to distinguish between; void and non-void return types. If the return type is non-void, returning a brace initialised object of type `Ret` will suffice for compilation, whereas void return types must be absent of non-void return values. Using `std::is_void_v<Ret>`, we can perform compile time type checking to circumvent this edge case and always ensure compilation for both void and non-void return types.

```
Ret branch(Args... args)
{
    if constexpr (!std::is_void_v<Ret>)
    {
        return Ret{};
    }
}
```

Now that the entry point is functional, we can double check the underlying assembly to ensure it has identical caller behavior to the branches. For this example, the branches used to instantiate the construct will be addition and subtraction functions with two integer arguments and an integer return type.

```
lea     rax, [rbp-16]
mov     edx, 2
mov     esi, 1
mov     rdi, rax
call    BranchChanger<int, int, int>::branch(int, int)
```

From the demangled function call it appears that the correct template is generated, however additional instructions have been added on the caller side. In addition to the integer arguments, an effective address is computed based on an offset from the frame pointer, which is moved into the `rdi` register after all previous arguments have been set up. From first glance it may be unclear why this is occurring, however when delving deeper into C++ calling conventions, what is happening is an implicit `this` pointer belonging to the specific `BranchChanger` instance is being pushed onto the stack [61]. If the `branch` member function utilised data members specific to the parent instance then this would be necessary, however this not the case and all it does is disrupt register offsets making trampolining to regular functions infeasible. Whilst it may be possible to rectify this programmatically using inline assembly, a safer and more portable solution would be to declare the `branch` method as static since static member functions are not associated with any particular instance of a class.

```
mov    esi, 2
mov    edi, 1
call   BranchChanger<int, int, int>::branch(int, int)
```

Static declarations allow the `branch` entry point to work seamlessly, however this limits the number of `BranchChanger` constructs that can be instantiated per function signature. Template specialisation allows for the differentiation of static `branch` entry points if the branches passed into the constructor have different signatures, allowing for multiple instances of semi-static conditions in a single program. However if more than one `BranchChanger` instance exists for a specific signature, this means they will share a common entry point which is problematic: two instances will be performing assembly modification on a single `branch` method which will have undefined behaviour. A way to circumvent is would be to alter the return types of the branches with other built-in or custom types to ensure that different templates are generated. Given these trade-offs, the priority for development is to work as much as possible with the compiler to avoid writing inline assembly for safety and portability reasons, and hence the final decision was to keep `branch` declarations as static.

Before moving onto implementing assembly modification, an important caveat to consider is compiler optimisations. From the compilers perspective, the `branch` method is a small function that does not produce any meaningful work making it susceptible to inlining or dead code elimination [62]. Obviously, if this happens then the construct will be unusable, but limiting its usage to programs that are intended to be run on **-O0** defeats the purpose of it being used in high performance applications. The simplest solution would be to disable optimisations specifically on the `branch` method, which can be achieved on GCC using `pragma` directives or more elegantly using attributes.

```
__attribute__((optimize("O0")))
static Ret branch(Args... args)
{
    if constexpr (!std::is_void_v<Ret>)
    {
        return Ret{};
    }
}
```

Assembly Editing Now that the entry point is fully functional, development can start for the core assembly modification functionality. The target for this editing will the prologue instructions of the `branch` method, specific to each instance produced by template specialisation of the parent `BranchChanger` class.

The first course of action is identifying the addresses of the machine code instructions to edit. In terms of the virtual address space, the machine code instructions of interest pertaining to the executable reside in the text segment, which itself is mapped by pages with read-only and execute permissions. Before we can perform any modifications, the page where the function prologue resides must be located

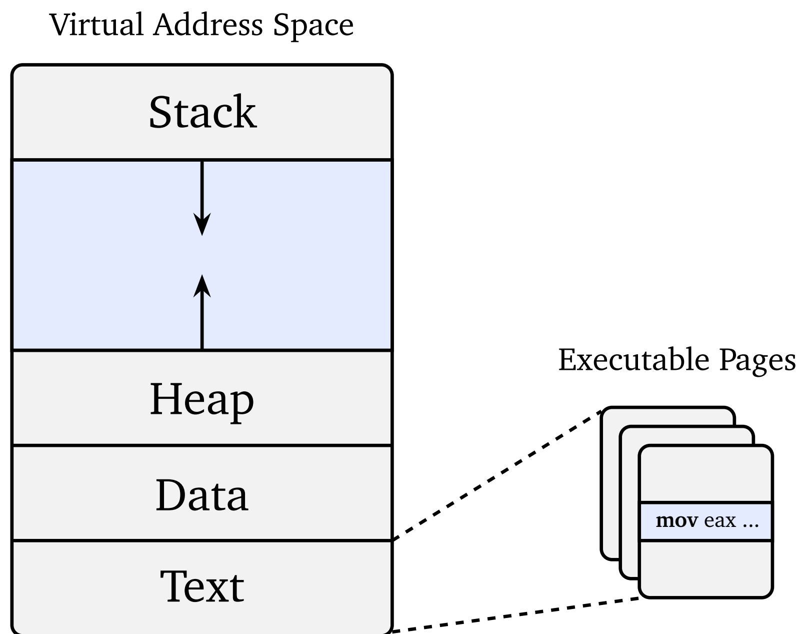


Figure 3.1: Simplified representation of virtual address space segments, with lower segments residing at lower memory addresses. Blue segment highlighted in executable page represents an offset where a specific instruction can be found.

and its permissions must be changed to read/write, otherwise the processor will raise a segmentation fault if any memory stores are attempted. Modern operating systems employ address space layout randomisation (ASLR) by randomising the base address of the virtual address space to prevent attackers from exploiting known memory addresses, so locating executable pages must be deferred to runtime [63]. Upon construct instantiation, we can generate a pointer to the template specialised `branch` method which is essentially the logical address of the first instruction pertaining to the function. To obtain the address of the page boundary in which the function resides, we can compute the page offset through the modulo of the logical address and the page size of the system (which can be obtained using the Linux specific `getpagesize` method), and then subtracting this offset from the original address to align it with the lowest multiple of the page size. To change the page permissions, we can use the `mprotect` system call to alter the flags of the VMA (virtual memory area, a kernel data structure which describes a continuous section in the processes memory) corresponding to the page address previously computed [64].

```
uint64_t page_size = getpagesize();
address -= (uint64_t)address % page_size;
mprotect(
    address, page_size, PROT_READ | PROT_WRITE | PROT_EXEC
);
```

Now that we have located the instructions in memory to edit (through the pointer to

the `branch` function), and made this editing permissible through altering the page permissions where the function resides, we can start adding instructions to redirect control flow to the branches. With latency in mind, the scope of control flow instructions that can be used become limited to direct jumps or calls, which conventionally cannot be polymorphic without employing assembly editing. On x86 architectures, jumps and calls redirect control flow by supplying a relative 32-bit offset from the current program counter, which is reduced to a simple signed displacement arithmetic operation.

```
00000000000011a9    <foo>:
00000000000011a9:   f3 0f 1e fa          endbr64
00000000000011ad:   55                   push   rbp
00000000000011ae:   48 89 e5             mov    rbp,rsp
(...)
00000000000011e3:   55                   push   rbp
00000000000011e4:   48 89 e5             mov    rbp,rsp
00000000000011e7:   e8 bd ff ff ff       call   11a9 <foo>
00000000000011ec:   b8 00 00 00 00       mov    eax,0x0
```

Above is an example of the machine code generated for a call instruction (achieved with **objdump -d -M intel**) along with the program counter (left) and equivalent assembly (right). Focusing on the call instruction with opcode **e8**, we can see the following 4-byte displacement encoded as a signed hexadecimal value in little endian format (architecture specific). The signed 2's compliment equivalent of this displacement is -67 bytes, which is 5 bytes smaller than the displacement from the PC of the call instruction (**11e7**) to the function entry point (**11a9**). The reason for this is because the offset of relative jumps are computed from the last byte of the instruction to the first byte of the target address, which can be formally described with the equation:

$$\text{Jump Offset} = \text{Target Address} - \text{Entry Point} - \text{Size of Instruction}$$

Fundamentally, relative jumps are indeed branches and introduce control hazards in pipelined processors, but have subtly different prediction schemes and penalties on the micro-architectural level in comparison to conditional jumps and indirect jumps (through register data). Prediction schemes for direct jumps occur relatively early in the pipeline (processor front-end), whereas branches that have data dependencies (such as indirect jumps/calls) or are conditional on FLAGS can get deep in the pipeline (back-end) execute stages before mispredictions are discovered, and hence incur a higher penalty when previous instructions need to be flushed. For relative jumps, the first line of prediction occurs in the branch target buffer (BTB) which acts as a specialised cache to predict weather the PC resolves to a branching instruction, and if so, what block to fetch next [17]. This is especially important for speculative prefetching: the instruction prefetcher needs to know in advance which blocks to fetch next, so if the PC is a branch, it can can steer the prefetcher to the predicted

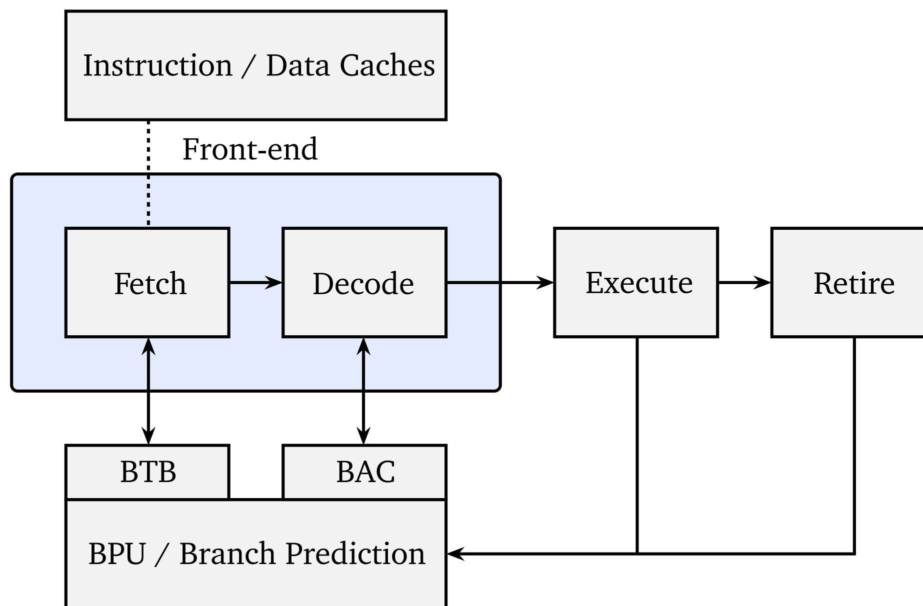


Figure 3.2: Simplified representation of the interaction of caches and branch prediction schemes on the instruction pipeline. Unconditional branch mispredictions are resolved at the decode stage by the BAC, whereas conditional branches are resolved at the execute stage. Information of retired predicted and mispredicted branches are fed into the BPU to update the predictor.

branch target and begin bringing the associated instructions into lower level caches. If predicted correctly then virtually no penalties are incurred as for conditional and data dependant branches, however the distinction occurs at the pipeline stage where mispredictions are detected and resolved. When an instruction reaches the decode stage, more information is attained surrounding the nature of the instruction. The branch address calculator (BAC) will ensure that the branches have the correct target by computing the absolute address of the PC and comparing it with the supplied target. If a direct jump is mispredicted at this stage, this means that the supplied branch target does not match the predicted, and proceeding instructions that have been incorrectly fetched will be flushed and the prefetcher will be re-steered to the correct branch target [65]. In regards to conditional branches, the same applies with the BAC however mispredictions are ultimately detected later in the execute stages which result in more severe pipeline stalls. On processors with branch order buffers (BOB), the recovery process can start before the processor pipeline has been flushed, but nevertheless the relative cost for mispredicted unconditional branches is much lower than conditional branches [65].

It is clear that relative jump/calls provide the cheapest means of control flow alteration, providing that the size of the jump does not exceed 2^{32} bytes. The question that remains is which instruction would be most suitable from a latency and implementation perspective. Call's and jumps are very similar mechanically, with calls being a two-part atomic operation which jumps to an offset while pushing the return address onto the stack. From a latency perspective, while minimal, the additional use

of the call stack pollutes data caches unnecessarily and the additional return instruction introduces more branching which is prone to mispredictions. However the main challenge comes from a development perspective; after the call is complete (within the `branch` method), the return address will be the proceeding instruction which may involve manipulating data on the stack/registers which have already been dealt with. Ensuring this does not occur for all possible signatures is tedious, and may limit future optimisations on the language construct. Using a jump will be much simpler; we can simply go straight to the branch without needing to ever complete execution of the entry point, and when the branch has finished executing, control flow will be redirected to the original calling site of the `branch` method (recall that a `ret` instruction is a `jmp [reg]`).

To implement the jump, the first thing we do is alter the opcode of the first instruction pertaining to the `branch` method to **e9** (`jmp` opcode on x86) through its pointer, and then increment it. The following 4 bytes will be reserved for the relative offsets from the current program counter. To compute the offsets, we simply use pointer arithmetic to compute the displacement in memory from the `branch` method to the respective if/else branches, then subtracting the length of the instruction from the offset as specified in the formula mentioned earlier. Next, the the integer offsets are converted into 4-byte hexadecimal representation and stored in a two dimensional member array (total of 8-bytes), accounting for architectural byte ordering.

```
unsigned char offset_in_bytes[DWORD] = {
    static_cast<unsigned char>(offset & 0xff),
    static_cast<unsigned char>((offset >> 8) & 0xff),
    static_cast<unsigned char>((offset >> 16) & 0xff),
    static_cast<unsigned char>((offset >> 24) & 0xff)
};

#if __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
change_byte_ordering(offset_in_bytes);
#endif

std::memcpy(dest_array, offset_in_bytes, DWORD);
```

Changing Branch Directions At this point the development of the construct is nearly complete, leaving only the direction changing method for development. With the offsets computed and stored in a class member array, setting the branch direction would simply involve a `memcpy` of these bytes to the `branch` pointer as the destination address. Direction setting must be initially done upon instantiation, since altering the first byte of the `branch` entry point will result in a jump to an undefined location, likely causing a segmentation fault. The class was adapted to have an optional parameter in the constructor to specify the initial direction of the branch, with the default condition being true. This can be conceptualised as a similar scheme to compiler branch prediction hints; the programmer can specify the likely direction which the branch would first be taken, but will still have the control to change

this at any given time. The actual `set_direction` method will use boolean indexing (the boolean being the runtime condition passed to the method) to access the bytes pertaining to the correct branch, which are copied into the 4 byte slot next to the `jmp` opcode. The boolean indexing approach is simple, and allows for active cache warming if desired.

```
void setDirection(bool condition)
{
    std::memcpy(dest, src[condition], DWORD);
}
```

Concluding Remarks Upon testing, the semi-static conditions prototype appears to work seamlessly for varying branch signatures on all optimisation levels. Whilst it is not possible to observe the assembly editing in real time without specialised disassemblers (for example, **objdump** only shows the contents of the object file which is not edited, rather the pages that are mapped by it), using **perf record** it is possible to observe it indirectly through the percentage cycles spent in the `branch` method.

```
Percent |
        |      BranchChanger<int, int, int>::branch(int, int)
100.00  |      push %rbp
        |      mov  %rsp,%rbp
        |      mov  %edi,-0x4(%rbp)
        |      mov  %esi,-0x8(%rbp)
        |      mov   $0x0,%eax
        |      pop  %rbp
        |      ret
```

Above shows the disassembly of the `branch` method with the percentage cycles spent on each instruction on the left hand side, obtained using **perf record**. The branches used in this example are simple addition and subtraction functions. The data shows that the first instruction within `branch` constitutes all the cycles spent in the function entirely; this is the instruction that is edited to a jump and hence it is expected that this is the only instruction that executes for the duration of the program. The branches themselves have a small percentage of cycles spent relative to all other methods in the test program, which supports that the branches are in fact executed and control flow is redirected accurately, this is also confirmed by simply printing the return values to the standard output. All the above, along with correct program behaviour, suggest that the language construct works as intended. Now the core prototype is complete, thought can be given into optimisations for branch taking and branch changing, as well as additional features that expand from this core concept (switch statements, also class member functions which observe different calling behaviour than conventional functions).

3.4 Optimisations

So far, the prototype demonstrates a proof-of-concept, but not a final product. The overarching goal of this language construct is to provide deterministic (low standard deviation) and low latency branch taking in scenarios where misprediction rates are high. Although mispredictions on the processor level are expensive, if the branch-taking component does not perform at a similar level to perfectly predicted branches, the construct will not find use in performance sensitive environments. Prior to running benchmarks, it is crucial that we level the playing field as much as possible to make semi-static conditions competitive.

Branch Taking Optimisations Naturally the most obvious place to start is the branch-taking method itself. In the development process, we ensured that the first instruction executed within `branch` is a jump that detours execution to one of the branches, so the processor does not waste any time executing instructions it does not need to. This is fine, however the glaring bottleneck arises from having to prevent all compiler optimisations on the method, which was implemented as a one-hot fix to prevent the function from being eliminated. Ideally, the entry point should benefit from all optimisations that regular functions do, but have the minimum amount of problematic optimisations disabled. The first obvious approach is do only disable in-lining for the entry point; this is destructive as the compiler will place the body of the function pre-editing within the calling site which essentially does nothing. Even if it managed to inline the edited assembly, it will be completely infeasible to target the in-lined instruction within the code segment. Replacing the compiler attribute on `branch` with `__attribute__((noinline))` generated the following assembly under **-O3** optimisations.

```
000000000000118c:    lea   rdx,[rip+0x26d]
0000000000001193:    lea   rax,[rip+0x136]
000000000000119a:    sub   rax,rdx
000000000000119d:    sub   rax,0x1
00000000000011a1:    mov   DWORD PTR [rip+0x25a],eax
```

Surprisingly, even with the `noinline` directive the compiler still reduced the `branch` call to the `lea` instructions highlighted in bold. Upon further research into the effects of GCC compiler attributes, what appears to be happening is that the inlining prevention does in fact take place, but the compiler deems the function to have no side effects and as a result optimises out the call completely. A simple way to control this optimisation is by inlining assembly within the function body; inline assembly adds uncertainty to the compilers optimiser as it cannot determine if it has side effects on register or memory values. Adding a simple `asm("")` which does not produce any meaningful work is sufficient to prevent optimising out the function call in addition to using the `noinline` attribute.

Further testing revealed an interesting yet problematic optimisation, calls to the original `branch` method where replaced with calls to a different `branch` method, of

which the compiler duplicated and altered a number of instructions within the body. Below is an example of both instances, with demangled function names simplified for readability.

```
0000000000001280: <_BranchChanger_branch.constprop.0.isra.0>
0000000000001280:     ret
0000000000001281:     cs nop WORD PTR [rax+rax*1+0x0]
0000000000001288:     nop
000000000000128b:     nop    DWORD PTR [rax+rax*1+0x0]
(...)
0000000000001290: <_BranchChanger_branch>
0000000000001290:     endbr64
0000000000001294:     xor    eax,eax
0000000000001296:     ret
```

The first example of the `branch` method represents the duplicated form which is called, whereas the second example represents the method which is needed to be called to allow semi-static conditions to work. Inspecting the demangled name of the duplicated function reveals the optimisation that has been applied: interprocedural constant propagation (ICP) [66]. This optimisation is multifaceted; when the compiler recognises that a function call has some arguments passed as constants, it creates a spot-optimised clone of the function which can involve removing redundant computations and memory accesses. This can be seen in the `constprop` version; the primary instruction becomes a `ret` because the compiler can see that the `branch` method does not produce any meaningful work, the remaining instructions are included as padding to align the function on a 16-byte boundary. This padding is important especially for procedural calls since most modern processors fetch instructions on aligned 16-32 byte boundaries; fetching code after after an unconditional jump costs a few clock cycles however this delay is worsened if the branch target does not lie on a 16-32 byte boundary [61]. Following this, an interesting observation can be made in regards to the original `branch` method. In many instances, the function itself does not follow alignment and as a result the compiler seems to always place it at the bottom of the text segment to prevent misalignment of all other procedures in the executable. Another interesting observation is that the `constprop` version is often placed close to hot code paths in the text segment (often very close to `main`), which can reduce instruction cache fragmentation by placing contiguous subroutines relatively close to one another.

The issue of preventing constant propagation and function cloning can be easily solved by including the `optimize("no-ipa-cp-clone")` attribute in the function header. However prior analysis into the effects of ICP and procedural reordering opens some interesting avenues into possible improvements. Taking a page out of the compilers book, the first observation of ICP was the reordering of instructions such that the most important instructions reside at the function entry point with no wasted work in between. In the improved `branch` method, the preliminary instruction is typically a 4-byte **endbr64** on Intel CPU's that employ control flow enforce-

ment technology (present on Linux with GCC and Clang), which ensures that indirect jumps/calls can only be made to functions which start in this instruction [67]. In the case of semi-static conditions, it is highly unlikely that the `branch` method will find use in indirect calls due to the associated costs with indirection in general, especially in low latency settings. Given this, overwriting this preliminary instruction with a 5-byte jump was the direction taken in development, however a key thing to note is that this editing overwrites more than one instruction owing to its greater length. While it is unclear what ramifications this has on variable instruction length pre-fetching, perhaps hard coding a 5-byte jump in the entry point (and editing will not alter the length of the instruction) will be more ”friendly” towards hardware semantics. In addition this may have positive implications on the BTB; from compile time the PC associated with the preliminary instruction of the `branch` method will always be a jump, meaning that in theory the BTB should always predict that a control flow instruction is present within `branch` which can save some cycles associated with mispredictions from preliminary calls. Even if later benchmarks reveal there is no observable performance gain from doing this, it does defer some work on construct instantiation.

To ensure that a unconditional jump always resides at the `branch` entry point, the **endbr64** instruction will need to be omitted by the compiler which can be done with the `nocf_check` attribute. Since there is already an assembly instruction present within `branch` to prevent the compiler from optimising out the call, this can be simply changed to `asm("jmp 0x0")` which hard-codes a jump to an arbitrary 4-byte offset, this will be edited exclusively. Upon inspecting the disassembly, the `branch` method starts to resemble its `constprop` counterpart even more:

```
0000000000001280: <_BranchChanger_branch>
0000000000001280:     jmp     0 <__abi_tag-0x38c>
0000000000001285:     xor     eax, eax
0000000000001287:     ret
```

Interestingly, this assembly ordering seems to be maintained regardless of the function signature; the compiler seems to understand to not optimise the function call so there is full benefit of caller setup and teardown, but it also understands that the function does no useful work and optimises accordingly. The only differences that remain now between the ICP counterpart is 16-byte alignment and procedural reordering. A simple way to ensure this is including the `hot` attribute which instructs the compiler to optimise the function more aggressively and places it in a subsection of the text segment where hot code lies. This is typically done automatically with the **–vprofile-use** flag to which the compiler uses profile feedback from previous executions to determine which functions can benefit from reordering. A caveat with using this approach is it relinquishes the programmers ability to decide which functions should have priority in the hot text segment, which may decrease performance depending on the application this is integrated in. However `hot` attributes are far more common across compilers than byte-alignment directives, so from a portability standpoint it would be easier to generate the desired assembly using this method.

Given these alterations, the final disassembly can be seen below:

```
0000000000001170: <_BranchChanger_branch>
0000000000001170:     jmp     0 <__abi_tag -0x38c>
0000000000001175:     xor     eax,eax
0000000000001177:     ret
0000000000001178:     nop     DWORD PTR [rax+rax*1+0x0]
000000000000117f:     nop
```

The altered version of the `branch` method, including the `hot` attribute is also shown below. Note that the `noinline` attribute is omitted since `no-ipa-cp-clone` includes this implicitly, and the function is always optimised on **-O3** to ensure the preliminary instruction is always a jump.

```
__attribute__
((hot, nocf_check, optimize("no-ipa-cp-clone", "O3")))
static Ret branch(Args... args)
{
    asm (jmp 0x0);
    if constexpr (!std::is_void_v<Ret>)
    {
        return Ret{};
    }
}
```

The improved `branch` method was benchmarked against the prototype version under various suites, broadly split into instruction-level benchmarks with inlined perf and Intel cycle counters, as well as micro-benchmarks with google benchmark involving measurements of more computationally expensive situations. More detailed methodology is explained in later sections, the purpose of these preliminary benchmarks is to ensure the proposed changes do not incur adverse effects on the language construct. Broadly on the instruction level, the optimised version improved performance by several cycles across different branches, with the most significant contribution coming from procedural reordering associated with the `hot` attribute. For higher level measurements, some benchmarks showed performance gain by 5-10% whereas others had identical execution times. On the instruction level it is difficult to speculate the source of this performance gain; at runtime both instances have identical execution pathways in terms of instructions so a reasonable explanation would be to improved locality between the entry point and branch targets. In larger systems with increased cache contention, the effects of alignment and locality have more of a prevalent effect on caching (branch method fits entirely in a single cache line) and prefetching which is reflected in some of the larger micro-benchmarks. Given these optimisations showed no adverse effects and showed marginal performance gain in some scenarios, they where incorporated into the final artefact.

Branch Changing Optimisations The current state of the direction changing method is already in quite an optimised state. There is an implicit branch for accessing the correct byte offset using a boolean index, however this is unavoidable. Fundamentally the performance of branch changing is not as important as branch taking; the whole reason for this separation is to isolate this more expensive operation from performance critical code to facilitate condition evaluation preemptively.

3.5 Generalisations

The focus of development has primarily been on semi-static conditions that emulate the behaviour of two-way conditional statements for conventional functions and static class member functions, and have been successful thus fair in exploiting calling conventions to facilitate safe branch-taking. Nevertheless, the current prototype has the capability to be further generalised to work for a larger scope of branches without needing to alter the core branch-changing logic. Whilst these extensions may not find as much use for the specialised case (branch optimisation in HFT environments), they will improve the flexibility of the language construct for more general use cases outside the field of low-latency development.

Class Member Functions The current state of semi-static conditions rely on standardised calling conventions to facilitate stack setup/teardown and function argument passing by the compiler, without needing to write inline assembly code. Before extending this to class member functions, one must examine the disassembly pertaining to these invocations to understand the necessary changes that need to be implemented.

```
0000000000002436:    lea     rax,[rbp-0x60]
000000000000243a:    mov     rdi,rax
000000000000243d:    call    263a <_ZN9SomeClass3fooEv>
0000000000002442:    lea     rax,[rbp-0x60]
0000000000002446:    mov     rdi,rax
0000000000002449:    call    268a <_ZN9SomeClass3barEv>
```

The underlying assembly shows similar behaviour encountered during the development of the `branch` entry point; the effective addresses being computed (highlighted in bold) represents an implicit `this` pointer to the parent instance which is the first parameter moved onto the stack. In this example both member functions are being invoked from the same instance, hence the identical offsets represented in the `lea` instruction. Propagating this behaviour to the entry point is simply the case of altering the class template to deduce the member function pointer type, and then updating the signature of the `branch` method to include the class instance within the signature prior to the parameter pack that represents the functions arguments.

```
template <typename Ret, typename Class, typename... Args>
class BranchChanger
{
    using func = Ret(Class::*)(Args...);
    ...
    __attribute__
    ((hot, nocf_check, optimize("no-ipa-cp-clone", "O3")))
    static Ret branch(const Class& instance, Args... args);
}
```

These alterations are sufficient for making semi-static conditions work for non-static member functions without needing to change any optimisations on `branch` or the core assembly editing logic. This will become a reoccurring theme in further generalisations. This extension is contrived to work only for member functions that belong to the same class, which is expected considering the approach used to deduce the class type through templating. Nevertheless multiple instances are able to share the same entry point and have their member functions invoked through `branch`, with support for derived class methods as long as they are not overloaded (if a derived class overloads a method from the base class and the base class pointers are passed to the constructor, only the base methods will be executed).

Switch Statements The design of the language construct make it seem that generalisation to *n*-ary conditional statements would be simple; simply change the template parameters and the offset storage array to reflect the number of branches. However the syntactic requirements complicate template deduction. If it was possible to alias parameter packs directly, this would be a simple task of deducing the function pointer signature from a variadic pack of pointers, extracting the return types and arguments externally and aliasing them from within the class. Unfortunately C++20 does not support this directly, and trying to work around this by using containers such as `std::tuple` to hold the arguments is non-trivial, since there is also the task of extracting these types and forwarding them to either a pointer or the `branch` template declaration.

The solution to this is to break the `BranchChanger` class into a base class and derived class. The base class will be partially specialised to extract the function signature (as seen so far), with the sole purpose of generating the `branch` method through template deduction and hence locating the bytes to edit.

```
template <typename T>
class branch_changer_aux {};

template <typename Ret, typename... Args>
class branch_changer_aux { ... };

template <typename Class, typename Ret, typename... Args>
class branch_changer_aux { ... };
```

The derived class will be the actual `BranchChanger` which the programmer will interact with. The class itself has variadic template parameters representing a number of function pointers with identical signatures, these will be the branches which can range from 2 to ∞. Using `std::common_type`, we can deduce the actual function signature type from the parameter pack which is used to instantiate the correct base class through CRTP.

```
template <typename... Funcs>
BranchChanger : public branch_changer_aux
<typename std::common_type<Funcs...>::type> { ... }
```

The only adaptation needed will be the constructor, which will need to expand the parameter pack and iterate over all pointers passed to the constructor, computing relative offsets and storing them element-wise. C++17 supports unpacking these types into into a `std::vector` directly using brace-initialised fold expressions:

```
BranchChanger(const Funcs... funcs)
{
    using ptr_t = typename std::common_type<Funcs...>::type
    std::vector<ptr_t> pack = { funcs... };
    for (int i = 0; i < pack.size(); i++) { ... }
    (...)
}
```

After a bit of hideous template meta-programming, the language construct becomes fully generalised to work for any number of branches, and both regular and member functions. Template deduction is completely abstracted from the programmer without the need of manually writing out types, providing and elegant and affluent interface for easy use and integration. This concludes the development of semi-static conditions, the remainder of development time was focused on productionising the construct into a library, with a focus on portability across different operating systems and compilers. This stage is rather dull and not worth discussing, most of the complexity arose from creating pre-processor macros to ensure different system calls and optimisations are enabled based on the users platform.

Chapter 4

Benchmarks and Applications

4.1 Outline

This chapter is dedicated to benchmarking the core operations that comprise semi-static conditions, exploring the effects self-modifying assembly instructions on performance, and investigating applications in both HFT and more general use cases. The experiments outlined will leverage a number of benchmarking suites ranging from Google Benchmark to custom performance counters that offer the fine granularity required to measure instruction level effects. The proceeding section is dedicated to outlining the experimental methods employed in obtaining these results for transparency and reproduciblity purposes. All measurements have been collected on an Intel(R) Core(TM) i7-10700 CPU (2.90GHz) with 256-kilobyte L1 instruction and data caches, 2-megabyte L2 caches and 16-megabyte L3 caches. Measurements collected will be architecture specific but have nevertheless been tested on similar architectures with Intel processors and have had consistent performance patterns with varying numbers. Tests have primarily been focused on semi-static conditions with regular or static member functions as branches given the large search space that exists with these kind of experiments.

4.2 Experimental Method

Conventional microbenchmarking frameworks such as Google Benchmark are useful for high level performance measurements where test cases are sufficiently long enough to measure observable differences in latency. However they often fall short for higher resolution measurements involving instruction level micro-benchmarks. When expected differences in performance manifest at the cycle level, the overhead associated with running these frameworks in conjunction with timer resolutions and standard errors mean that any observable differences in latency are *hidden* by background noise, and often results become more influenced by the measurement taking rather than the actual measurements. This section is dedicated to outlining the experimental methods employed in capturing these sensitive measurements, based heavily on work done by Agner Fog, Matt Godbolt, and Intel as part of their microbenchmarking guides for i7 processors [68, 69, 70].

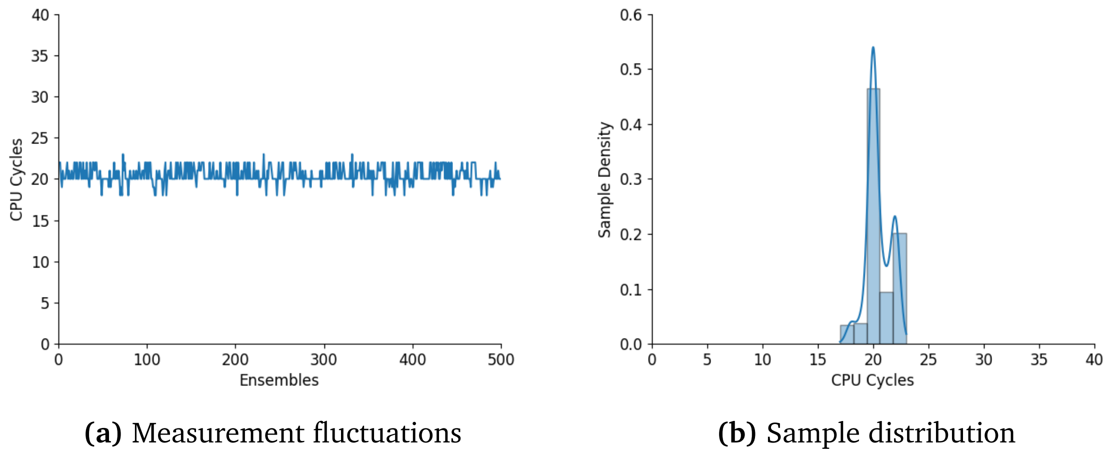
Clock Cycle Measurements Benchmarks for code comprised of small numbers of assembly instructions where conducted using architecture specific timestamp counters, in this case RDTSC was used. Measurements are taken by reading the processors timestamp counter at two intervals with the code to benchmark in between, it is important to note that RDTSC counts reference cycles rather than core clock cycles due to CPU throttling effects. On super-scalar processors instructions are executed out-of-order and in-parallel to optimise penalties associated with different instruction latencies. This is problematic for such measurements; there is no guarantee that the RDTSC instruction is called in the precise temporal order that is specified programtically, and measurements may include other assembly instructions that are not intended to be measured. To resolve this, serialising instructions can be used in conjunction with RDTSC to force the CPU to complete all preceding instructions before continuing execution. Examples of serialising instructions are CPUID and LFENCE, in all tests LFENCE is used as it has a lower overhead and does not clobber the output registers of RDTSC. An example setup can be seen below, this snippet has been adapted from the official Intel microbenchmarking guide for i7 processors [70].

```
_mm_lfence();
uint64_t start = __rdtsc();
_mm_lfence();

code_to_measure();

_mm_lfence();
uint64_t end = __rdtsc();
_mm_lfence();
uint64_t cycles = end - start;
```

Compiler optimisations often reorder assembly which complicates measurement taking. While there is no set way to ensure this, a trial and error approach was taken by padding instructions before measurement taking and cross checking the disassembly to see if the necessary code is placed between the RDTSC calls.

Measurement taking itself does incur some overhead. To account for this, prior to benchmarking a background measurement is taken by running the above code with no instructions in between the RDTSC calls for many iterations (often in the order of 10^7), from which a mean latency is computed and subtracted from all proceeding benchmarks (excluding outliers). The benchmarks themselves are also run for many iterations since measurements tend to fluctuate around a mean value after sufficient warm-up time (examples shown in Figure 4.1), due to variance in CPU frequency and individual instruction latencies. Therefore, data collected for benchmarks are processed as distributions rather than fixed computed values, which is beneficial for reasoning about latency standard deviations (important for HFT) and observing hardware level effects that contribute to this (e.g., branch mispredictions).

Profiling CPU performance counters are incredibly useful in identifying sources for



(a) Measurement fluctuations **(b)** Sample distribution

Figure 4.1: CPU cycle measurements of RDTSC and LFENCE overhead.

particular hot-spots during program execution, and in the context of this research, identifying granular hardware effects that contribute to observed latencies. Perf was primarily used for performance profiling. A downside to this is that perf traditionally profiles the entire executable, rather than small subsets of it, meaning that any small observable changes in hardware counters that are expected become enveloped in the overall noise of the system. Luckily, Linux offers a API to access a subset of perf performance counters inside the executable using the `perf_event_open` system call, allowing for small pieces of code to be profiled in isolation (similar to RDTSC) [71]. Events are set up using the following code:

```
struct perf_event_attr attr;
attr.type = PERF_TYPE_HARDWARE;
attr.config = PERF_COUNT_HW_INSTRUCTIONS;
attr.disabled = 0;
attr.exclude_kernel = 1;
attr.exclude_idle = 1;
attr.exclude_hv = 1;
attr.exclude_guest = 1;
```

The `type` and `config` attributes are used to select the performance counters which are generally the only parameters that are changed between tests. The remaining attributes offer finer control over sampling and are configured to exclude external noise from measurement taking. The actual profiling code can be shown below which has been adapted from the Linux documentation of `perf_event` [71]:

```
fd = perf_event_open(&attr, getpid(), -1, -1, 0);
ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
```

```
code_to_profile();

ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
rc = read(fd, &count, sizeof(count));
```

Whilst this approach offers the best solution to granular profiling, the drawback is that the `perf_event` API only offers a small subset of performance counters that perf offers. In experiments that utilise profiling, this inline approach is used when applicable, whilst the command line approach is used when performance counters that cannot be obtained using `perf_event` are needed. In this case, code is often kept to a minimum to reduce measurement noise and often run alongside a baseline to extrapolate differences in performance counters.

Microbenchmarking Frameworks Where applicable, Google benchmark was used to gather latency data for larger events. Google benchmark automatically configures the number of iterations the benchmark is run to get a stable estimate [72].

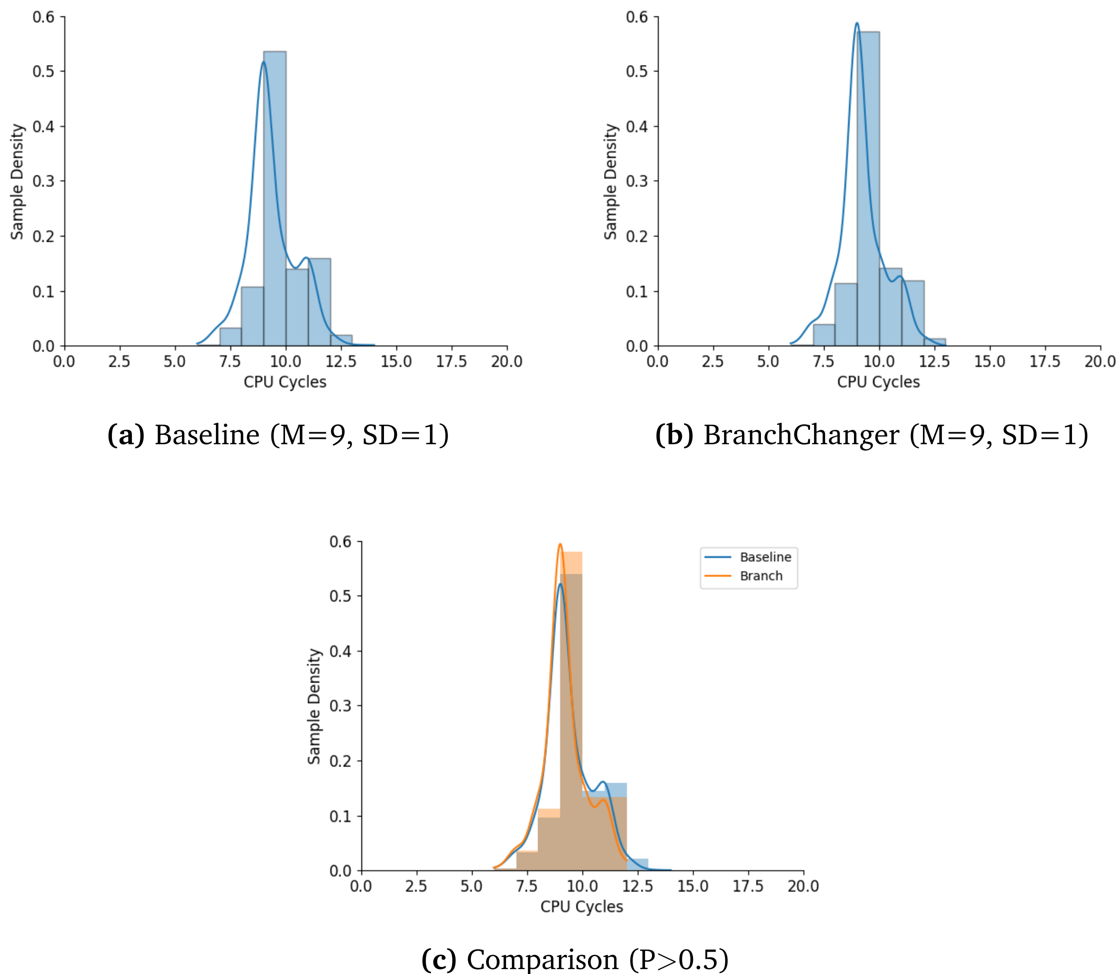
4.3 Benchmarks

This section is dedicated to benchmarking the branch-changing (`set_direction`) and branch-taking (`branch`) methods of semi-static conditions, exploring the effects of self-modifying code and deducing optimal usage. Often, measurement distributions do not follow standard distributions due to skewness, so non-parametric tests are conducted on small subsets of the samples where applicable.

Branch-changing Benchmarks This set of tests is concerned with exploring the instances where altering assembly instructions in memory cause performance degradation and how they can be avoided.

The first test benchmarks the performance of `set_direction` versus an equivalent 4-byte `memcpy` to non-executable memory. For fairness, a class was created with identical data members to semi-static conditions which where initialised with random bytes to represent some runtime deduced data. The `set_direction` method in the baseline class is identical to the one in `BranchChanger`.

```
class Baseline
{
private:
    unsigned char* bytecode;
    unsigned char bytes[2][DWORD];

public:
    Baseline()
    {
```



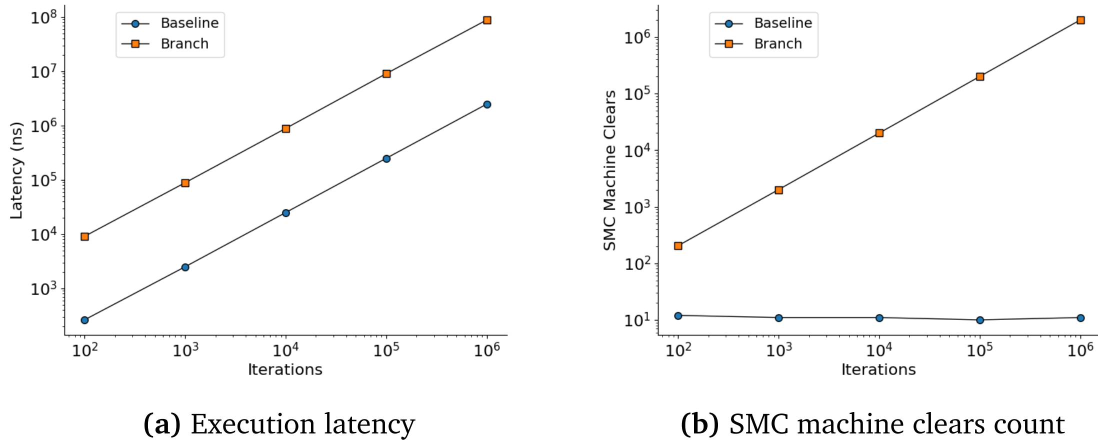
(a) Baseline (M=9, SD=1) **(b)** BranchChanger (M=9, SD=1)

(c) Comparison (P>0.5)

Figure 4.2: Benchmark results in CPU cycles for branch-changing overhead versus an equivalent 4-byte `memcpy` to non-executable memory.

```
        bytecode = new unsigned char[DWORD];
        _random_bytes(bytes[0]);
        _random_bytes(bytes[1]);
    }

    void set_direction(bool condition)
    {
        std::memcpy(bytecode, bytes[condition], DWORD);
    }
};
```

Interestingly, writing to executable memory on its own does not incur any additional penalties with respect to the baseline, which is reflected by the near identical distributions in execution latencies shown in Figures 4.2(a)-(c). It is understood that modern processors tolerate self-modifying code (SMC) but are in no way friendly to-



(a) Execution latency

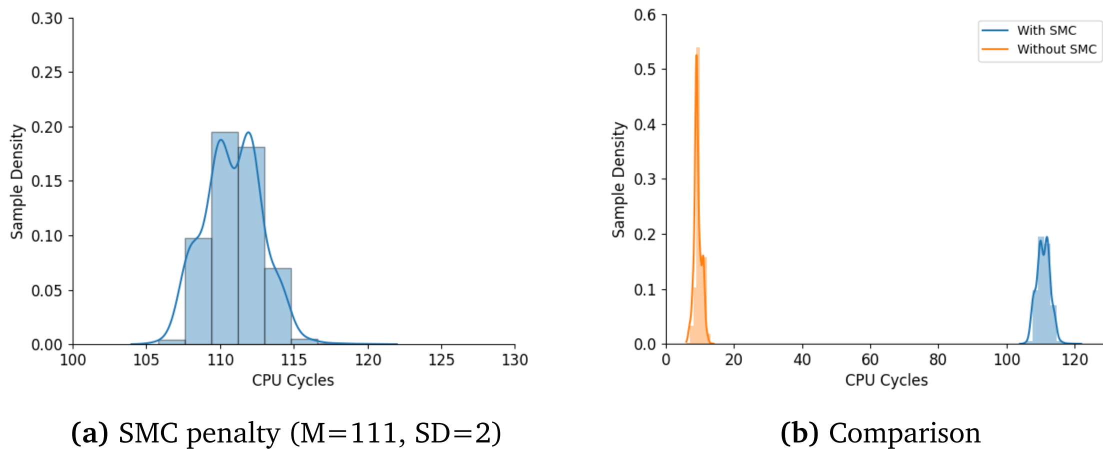
(b) SMC machine clears count

Figure 4.3: Latency and SMC machine clear count measurements for branch-changing followed by immediate branch-taking (labelled branch) using semi-static conditions.

wards it, often initiating full pipeline and trace cache clears which can cause penalties in the hundreds of cycles [73]. The actual semantics of how processors detect SMC is unclear, however the general consensus in architectural forums and patents point towards a "snooping" mechanism which is initiated by store instructions to executable memory addresses. These snoops compare physical addresses of in-flight store instructions with entries in instruction cache-lines to see if the store location corresponds to instructions in executable memory. If there is an address match, the SMC clear is initiated and new instructions are fetched from memory to lower level instruction caches [74]. Understanding this the results make sense; since there is no branch-taking occurring (where SMC is executed) there are no traces of the altered instructions in instruction cache lines, pre-fetch queues, or the i-TLB and hence the physical address check fails to initiate SMC clears.

Following these observations, the next test involved benchmarking the semi-static conditions `set_direction` method followed by branch-taking, with the baseline having `branch` replaced with a direct call to one of the functions passed to the constructor. The goal is to try and trigger SMC machine clears following previous discussions, and measure their associated penalties.

```
__attribute__((optimize("O0")))
void if_branch() { return; }

__attribute__((optimize("O0")))
void else_branch() { return; }
(...)
for (int i = 0; i < iterations; i++)
{
    branch.set_direction(condition);
    branch.branch()
}
```



(a) SMC penalty (M=111, SD=2) **(b)** Comparison

Figure 4.4: CPU cycle measurements for branch-changing with SMC machine clear penalty.

As expected, the presence of branch-taking close to the branch-changing method triggers large numbers of SMC machine clears. The penalty of this is quite severe; extrapolating the execution latencies from Figure 4.3(a) reveal that SMC machine clears multiply running times by 30-40x in this benchmark, with approximately 2 clears occurring per iteration on average (from Figure 4.3(b)). The additional overhead of SMC machine clears seems to be approximately 100 cycles (Figures 4.4(a)-(b)) which is consistent with approximations made by Agner and Intel, and are in agreement with various benchmarks made on architectural forums [68]. Nevertheless the machine clear trigger seems to have deterministic behaviour; Figure 4.3(b) shows that the number of SMC clears scale linearly with iterations, which is reflected in the overall execution latency. It can be said for certain that executing modified assembly instructions relatively soon after editing initiate these clears, which outlaws the use of branch-changing inside tight loops when conditions change frequently.

Whilst the cost of such machine clears and their trigger are easy to reason about and are deterministic, the actual segments of assembly where these penalties manifest are not. This is problematic; the total cost of the branch-changing method is potentially propagated to areas of code outside of itself in the form of SMC penalties, which introduces uncertainty in execution latencies for code that may be performance critical (such as `branch!`). Ideally, there should be some large enough buffer within the branch-changing method to contain this cost exclusively within `set_direction` for more deterministic execution latencies.

An interesting observation is that there seems to a ”lag” period from when the `set_direction` method is executed to where the SMC cost starts to manifest, assuming that instructions are executed in the temporal order in which they appear in the disassembly. This is consistent with behaviour observed by Ragab et al.; the process of initiating the pipeline snoop to triggering the SMC clear takes several cycles since it involves instruction stream walks and i-TLB checks, resulting in a transient execution window of stale instructions caused by the de-synchronisation of the store buffer and instruction queue [75]. From a prevention standpoint, strong serialis-

ing instructions such as CPUID and SERIALISE where inserted into set_direction to see if they where capable of preventing the SMC trigger, however this was not the case. In previous discussions we established that SMC triggers are caused by comparisons of in-flight or executed store instructions with physical addresses in instruction caches, so the ineffectiveness of serialising instructions is clear. Whilst CPUID and SERIALISE force the processor to complete all previous instructions and even drain the store buffer to prevent reordering, they do not have influence on instruction caches from which SMC checks are conducted, further supporting the presence of such ”snooping” mechanism.

It is possible to manually flush instruction cache lines pertaining to branch using the _mm_clflush intrinsic which shows some promise in minimising SMC clears when assembly modification is temporally closer to branch-taking, reducing such clears to approximately one per edit. Locality in this sense is very important; the closer assembly editing is to branch-taking, the more prominent the effect of SMC clears since the stale instructions have now polluted the pipeline, caches and instruction queues and thus require more machine clears to rectify. In the event that branch-changing occurs right before branch-taking, cache flushes do not seem to have a net positive effect which supports the notion that locality (in terms of instructions queued from the point to modification to the point where the modified code is executed) of SMC is the determining factor for the severity of associated penalties. Interestingly, Intel's optimization manuals do in fact recommend that SMC should not share the same 1-2 kilobyte sub-page for speculative prefetching and execution reasons, further supporting this notion [73]. To test this hypothesis further, an artificial buffer was created which comprised of a cache flush followed by some computational work, acting as a temporal barrier between assembly modification and branch taking.

```
__attribute__((optimize("O0")))
void _smc_buffer()
{
    _mm_clflush(address_of_branch);
    uint64_t buffer[DWORD * 4];
    for (int i = 0; i < DWORD * 4; i++)
        buffer[i]++;
}
```

Executing the following code directly after assembly modification is indeed sufficient in halving SMC clears, the same behaviour observed when adding cache flushes relatively close to branch-taking. The cache flush prevents additional machine clears due to physical address matches between store instructions and instruction cache data, whilst the computational work provides a sufficient buffer within the instruction pipeline to prevent similar matches with in-flight instructions. Usage of such a buffer will be optional to the programmer, but would also require some sort of active cache warming with branch to ensure that memory access penalties do not propagate to the hot-path. A simple optimisation to minimise clears would be to conditionally check if condition passed to set_direction is the same to the current

direction already set; in the current state, assembly modification is performed even when it isn't needed and always initiates machine clears. As it is not possible to completely remove SMC penalties, **any overhead ought to be isolated within the branch-changing method**. This was achieved by forcing the execution of the `ret` instruction within the `branch` method through casting the associated byte-code to a void pointer and calling it within `set_direction`. By executing part of the `branch` method, the modified instructions are brought sufficiently close to the CPU such that the snooping mechanism can quickly identify the modified instructions and initiate the pipeline clears **exclusively** within `set_direction`, offering a more deterministic performance model.

Branch-taking Benchmarks This set of tests is concerned with comparing the efficiency of the branch-taking method with conventional direct function calls, and exploring its implications on the BTB.

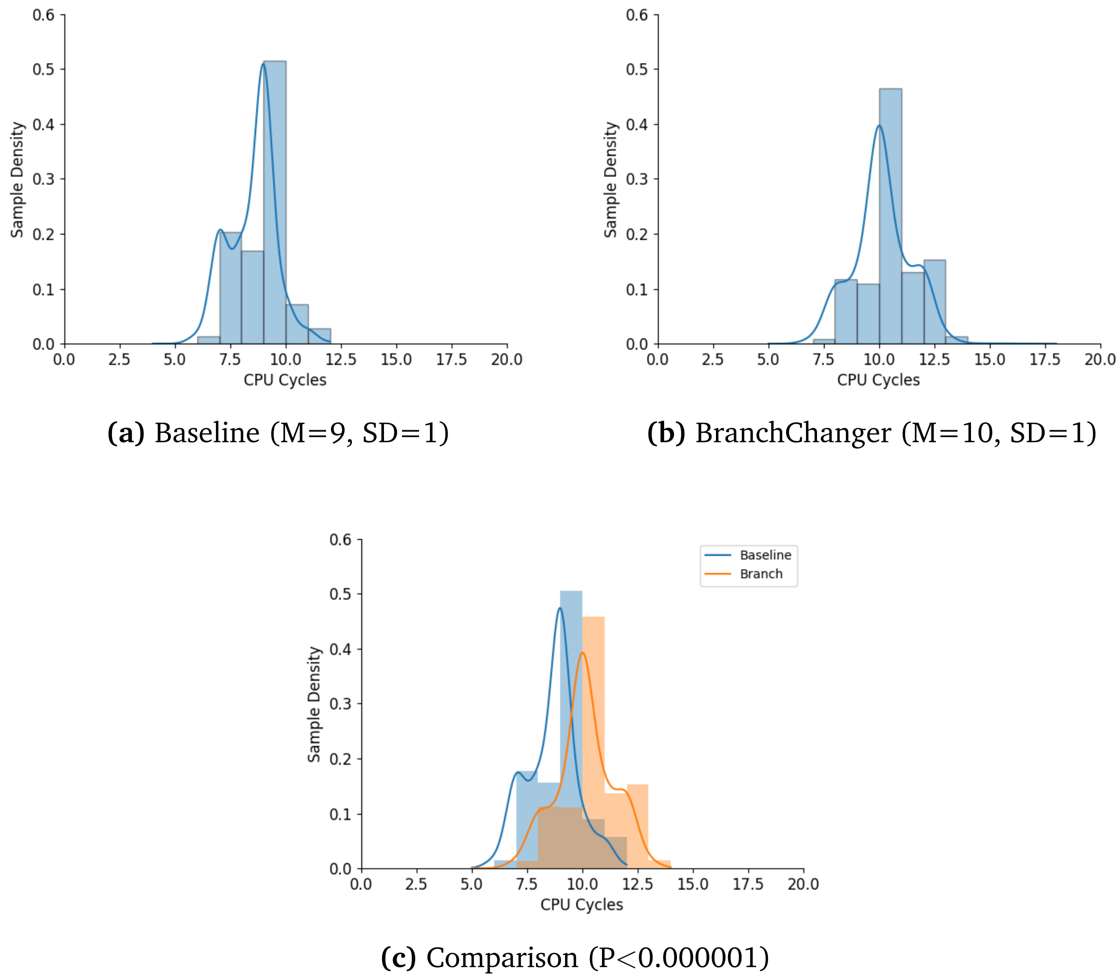
The first test investigated the overhead of `branch` versus a direct function call. The function call latency served as the baseline and was the same function that was executed through the `branch` entry point, so the branch direction was never changed.

```
__attribute__((optimize("O0")))
void if_branch() { return; }

__attribute__((optimize("O0")))
void else_branch() { return; }
```

Measurements where taken in the following manner, repeated for 10^7 iterations.

```
void measurement()
{
    start_measurement();
    branch.branch();
    stop_measurement();
}
```

Under the same conditions branch-taking has virtually identical overhead to a regular function call, with equal standard deviations in execution latency as seen in Figures 4.5(a) and 4.5(b). The small difference in overhead may be attributed to the additional `jmp` instruction that resides within the prologue of `branch`, which is the only difference between the execution pathways of the baseline and semi-static conditions. An experiment to measure the latency of a single relative `jmp` on this specific architecture would validate the hypothesis, however this is tedious and unnecessary. Using Agner's instruction benchmarks for Intel and AMD CPU's (note that the CPU that the benchmarks where run on is part of the 10th generation Comet Lake family, this was unavailable in Agner's instruction tables so reference latencies



(a) Baseline (M=9, SD=1)

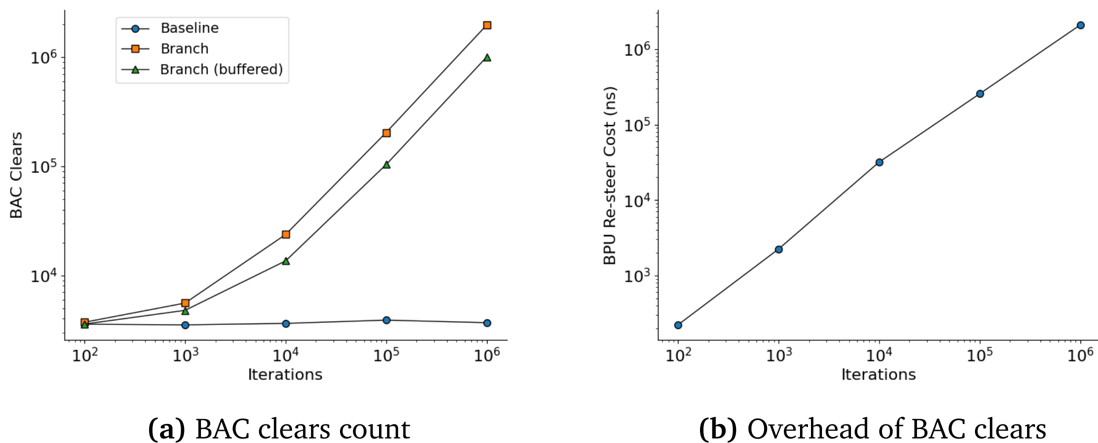
(b) BranchChanger (M=10, SD=1)

(c) Comparison (P<0.000001)

Figure 4.5: Benchmark results in CPU cycles for branch-taking overhead versus a conventional direct function call.

from Ice Lake where used), the typical latency of a relative jump is 1-2 cycles which falls within the range of the observed differences [76]. Given the large number of iterations that the benchmark was run and that the branch direction was never changed, it is likely that the instructions that where measured where hot in lower-level caches with optimal usage of other hardware semantics, skewing the latency of the additional `jmp` towards the minimum value, and thus supporting the hypothesis.

The implications of these results are multifaceted. The current implementation of branch-taking seems to be optimised to the theoretical limit; the additional overhead incurred seems to be caused exclusively by the additional control flow instruction that is inserted, which fulfills the overarching goal for its implementation. In low-latency environments such as HFT, low standard deviations are important for deterministic execution times and hence are a focus in development. The branch-taking method has shown that it has virtually identical standard deviations in execution latency as an isolated function call, offering programmers the assurance of determinism with respect to the current state-of-the-art.



(a) BAC clears count

(b) Overhead of BAC clears

Figure 4.6: BAC clear counters for continuously changing branch targets (branch) versus static branch targets (baseline, `set_direction` is always true), total overhead is calculated by subtracting the baseline latency from the benchmark. Branch (buffered) in (a) represents some computational work between `set_direction` and `branch`.

In the development portion of the report, we discussed the difference in prediction schemes for conditional and unconditional branches. In theory, when the branch direction is changed, the BTB entry corresponding the jump instruction within the `branch` prologue becomes invalidated due to an incorrect branch target. Whilst the BTB will still accurately predict if the PC is indeed a jump, the stale branch target will likely be detected by the BAC which will initiate a pipeline flush and re-steer the prefetcher. To observe this behavior, we run the following testing suite with **perf stat -e baclears.any:u** which counts the number of BPU front-end re-steers initiated by user-space code:

```
for (int i = 0; i < iterations; i++)
{
    branch.set_direction(condition);
    branch.branch();
    condition = !condition;
}
```

Similar to SMC machine clears, BAC corrections appear to increase linearly with iterations when the branch direction is continuously changed in deterministic fashion. An interesting observation in Figure 4.6(a) is that introducing a buffer of computation between branch-changing and branch-taking calls halves the number of corrections, averaging one clear per iteration. This suggests that Intel BTB's are updated atomically upon the retirement of branch macro-instructions; without the buffer the measurement loop is sufficiently small in terms of computation such that the modified `jmp` from the next iteration enters the pipeline before the current `jmp` gets retired. Since the BAC does not update BTB entries, but rather re-steers the

prefetcher to the correct branch target, it initiates two re-steers per iteration due to stale BTB entries. The penalty of the correction also scales linearly, averaging an additional 2.2 ns per iteration which equates to approximately **6 cycles** (likely overestimated) on this particular architecture, less than half the cost of a conditional branch misprediction (presumed around 13 cycles on Skylake CPU's) [76].

Though less severe, branch-taking using semi-static conditions can in-fact impose minor misprediction penalties, however it can be mitigated from the hot-path. The misprediction is essentially a one time cost when switching branch-directions; once the BTB has been corrected with the updated branch target, all successive calls `branch` will incur zero penalties and be executed with minimal latencies. This allows programmer to do effective 'warming' in the cold-path which is not possible with conditional statements; branch prediction is facilitated through capturing histories of branches based on their program counter and correlating them with global patterns. Adding conditional statements in the cold path in an attempt to 'warm' the BPU for the identical hot-path code will likely have little effect since the BPU treats both branches separately based on PC. Whilst it may capture some correlation, introducing more branches pollutes the global history can also introduce more noise. Conversely, calling `branch` in the cold path to warm the BTB works since control flow is redirected to the PC with the stale `jmp`, facilitating the correction preemptively and also brings the associated branch target (the functions being branched to) into lower level instruction caches. Using HFT as an example, this can be realised by sending 'dummy orders' through the `branch` method after the branch-direction has been set to ensure warming occurs.

```
void cold_path()
{
    (...)
    do_some_work();
    branch.set_direction(condition);
    branch.branch(dummy_order);
    do_some_more_work();
    (...)
}
```

Interim conclusions Results obtained from the preceding test suite provide valuable insight into the implications of using semi-static conditions on the hardware level, which in turn help deduce optimal usage. The majority of the cost with using this language construct is manifested within the `branch_changer` method in terms of machine clears caused by self modifying code, whilst branch-taking (the latency critical part) has extremely low overhead with deterministic execution latencies. In the preliminary sections of the report, it was hypothesised that optimal usage would involve separating branch-changing and branch-taking into cold and hot paths respectively, providing a means for pre-empative condition evaluation and branchless execution. The benchmarking results validate this use case, showing immense promise for branch optimisation in code paths that are infrequently executed, but

contain branches that are poorly predicted. In addition, the tests brought novel investigations into modern Intel processor semantics, exploring behaviour relating to SMC and branch-prediction pertaining to unconditional branches, which in turn can help low-latency developers optimise their code paths respectively.

4.4 Applications

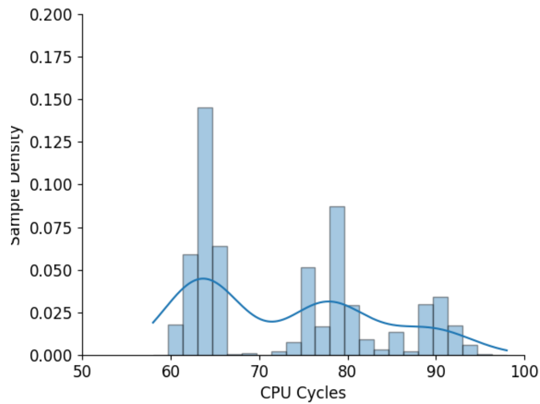
This section builds off the findings of Section 4.3, exploring the effect of removing runtime checks through branch-taking with semi-static conditions and how its performance compares to current state of the art: conditional statements with branch prediction.

Hot-path optimisation in HFT In contrary to traditional engineering terminology, the "hot-path" in the context of HFT refers to a section of code which is executed relatively infrequently, but when it is executed it needs to be extremely fast. This is typically the time taken from receiving market data to sending an order, with the whole process taking just several microseconds to execute. Lying at the heart of the trading system, and probably the most critical piece of code in terms profitability, this area is the target for most optimisation. Since it executed relatively infrequently, the remainder of the trading system is often designed with cache warming measures that keep data used in the critical path hot in low level caches to avoid memory access penalties. However this poses a challenge for branch prediction; branches in the hot path may have limited histories with complex patterns, resulting in mispredictions. Here, semi-static conditions are applied to optimise **branch-taking** for this use case.

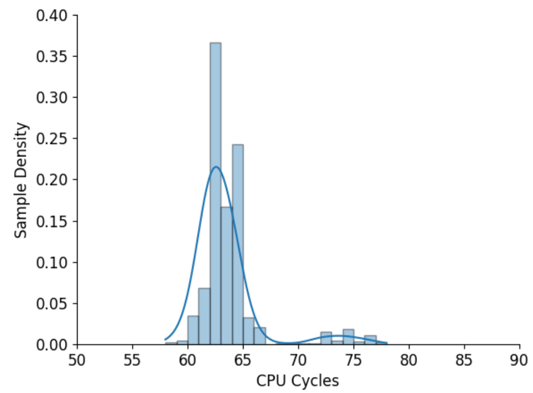
The test suite comprised of a tight measurement loop using RDTSC counters where runtime conditions where randomly generated using the Mersenne Twister Engine. For an infinite series of randomly generated booleans, it is expected that roughly half will be true and false, however since measurements are finite there will likely be skews towards one boolean in benchmarks. The choice of branches to benchmark initially where 64 byte memory copies and bit flips to a volatile `struct`, the scenario representing message passing to custom firmware (such as network cards and FPGA's) in a HFT system.

```
__attribute__((noinline))
void send_order(unsigned char* message)
{
    std::copy(message, message + 64, FPGA.payload);
    FPGA.flag = !FPGA.flag;
}
```

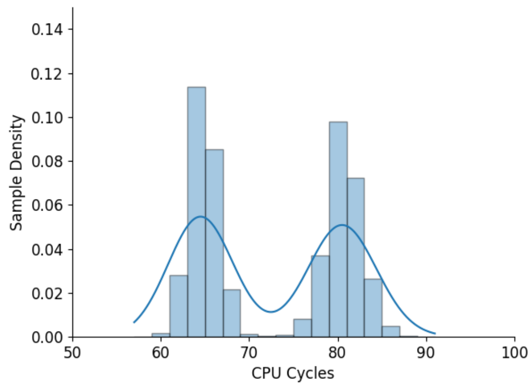
Emulating the hot-path in terms of infrequent execution is especially challenging from a benchmarking point of view. Percentage cycles spent in the measurement zone (the hot-path) where used as a proxy, and this was minimised by adding com-



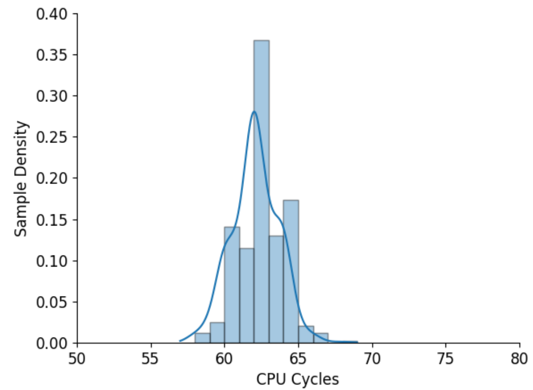
(a) Conditional statements without cache warming (M=75, SD=10)



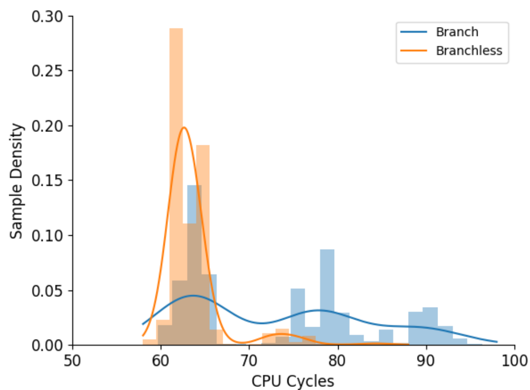
(b) Semi-static conditions without cache warming (M=63, SD=3)



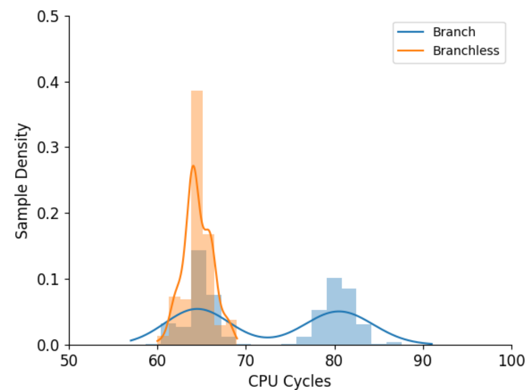
(c) Conditional statements with cache warming (M=68, SD=8)



(d) Semi-static conditions with cache warming (M=62, SD=2)



(e) Comparison without cache warming



(f) Comparison with cache warming

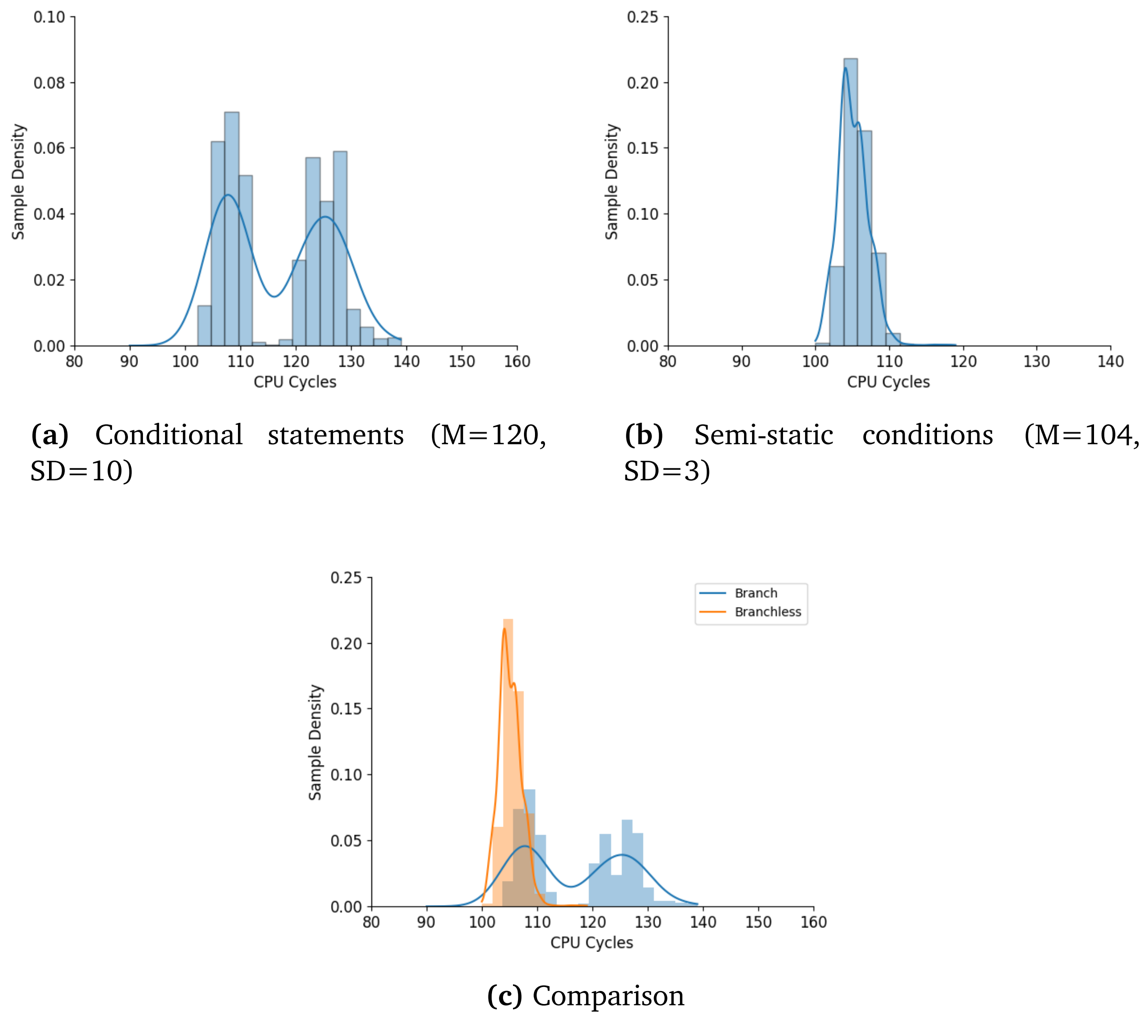
Figure 4.7: CPU cycle measurements of conditional branching (branch) versus semi-static conditions (branchless) with and without cache warming.

putational work in the form of randomly generating messages and running pricing calculations outside the hot-path. The randomly generated messages where passed to the branches in attempt to emulate real-time message passing, and measurements where taken with and without cache warming.

The results shown in Figure 4.7 are quite remarkable. Across the board semi-static conditions produce a tight uni-modal distribution with median latencies and standard deviations much lower than conditional branching, an artefact of removing runtime checks when needing to execute the branch. In terms of conditional branches, the sample densities elegantly model the latencies of predicted and mispredicted branches as a bimodal distribution which contribute to the larger median and standard deviation in cycles. In tests without cache warming, both conditional branching and semi-static conditions have a small distribution of measurements in the 70-90 cycle range presumably due to cache effects caused by message passing, which seems to disappear completely when cache warming is employed. Looking closer at the bimodal nature of conditional statement execution latencies (M=65 SD=2, M=78 SD=2 without cache warming and M=64 SD=2, M=80 SD=2 with cache warming respectively), the difference in median values between both distributions is 13-16 cycles which is in good agreement with misprediction penalty estimations made by Agner for i7 processors [68]. Through using semi-static conditions, it is clear that **runtime checks are entirely removed** during branch execution. Programmers gain the benefit of branch-taking latencies comparable (in this case even slightly better) to perfectly predicted branches, saving 2-4 ns on average and up to 6 ns if the branch is always mispredicted, with more deterministic execution latencies manifesting as low standard deviations.

Subsequent tests with other branches reveal the same behaviour for both conditional statements and semi-static conditions; whilst the distributions where centred around different medians due to the varying execution time of different branches, the relative offsets between the distributions remained the same. Interestingly usage of the `[[likely]]` and `[[unlikely]]` branch prediction hints had no effect in mitigating the misprediction rate for conditional branching. The reason for this is clear; compiler hints simply reorganise the assembly code to aid the processors static predictor in taking the more likely execution path, but since conditions are random and neither path is more likely being taken it has no net positive effect. That being said, it does seem that semi-static conditions are better than branch prediction hints for this use case. Further tests added more computational work to the hot path to observe this behaviour when branching is surrounded by more complex logic, an example of the measuring suite for conditional branches is seen below.

```
void measure_hot_path()
{
    begin_measurement();
    (...)
    do_some_calculations();
    if (condition)
        send_order(message);
```



(a) Conditional statements (M=120, SD=10)

(b) Semi-static conditions (M=104, SD=3)

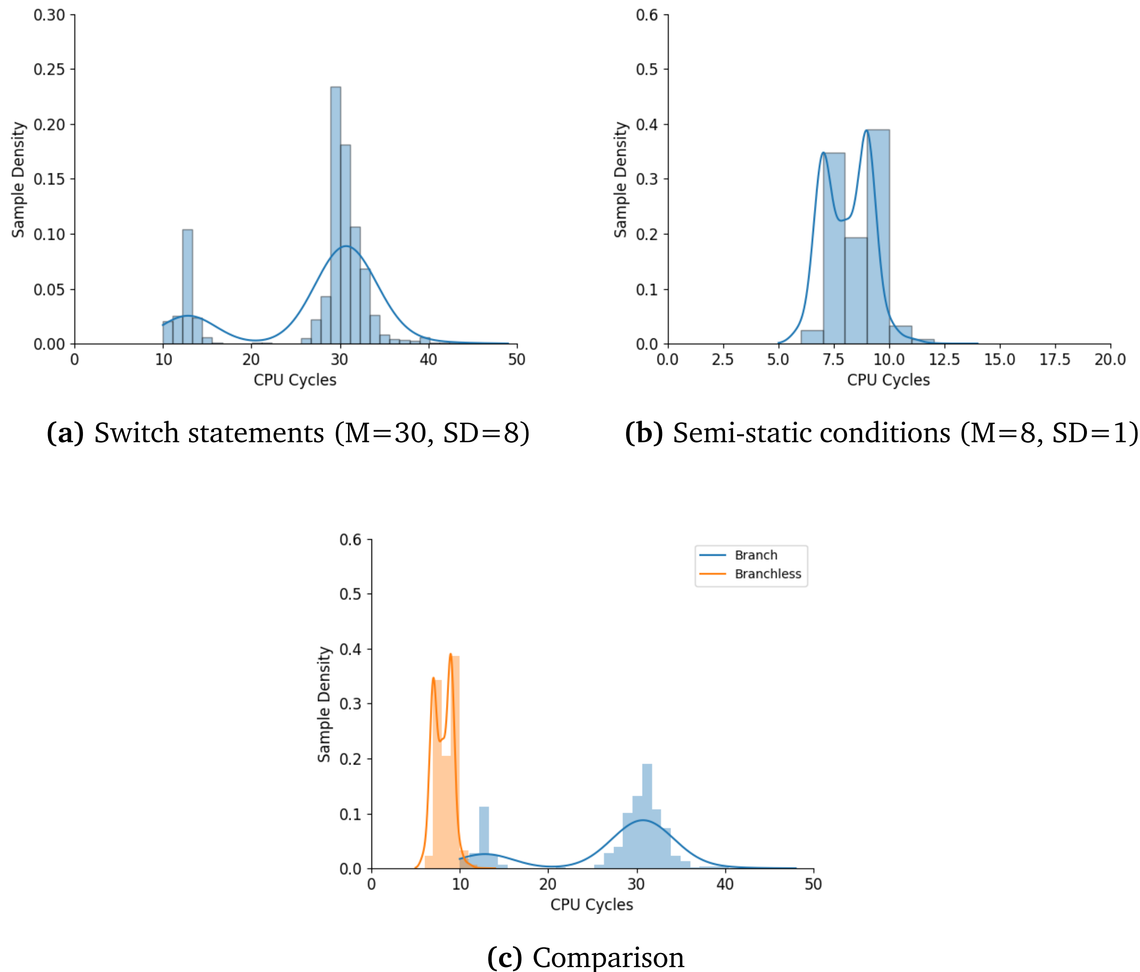
(c) Comparison

Figure 4.8: CPU cycle measurements for conditional branching versus semi-static conditions with updated hot-path.

```
    else
        adjust_order(message);
    Or branch.branch(message)
    do_some_work();
    (...)
    end_measurement();
}
```

Cycle distributions in Figure 4.8 seem to follow an identical pattern to Figure 4.7 regardless of the additional computational work. It seems that misprediction cost for this case increased slightly for conditional branches to 18 cycles (M=108 SD=2 for predicted and M=126 SD=4 for mispredicted) which suggested that branch mispredictions can impact surrounding code resulting in higher penalties.

In the case of *n*-ary conditional statements in the form of if-else chains or switch



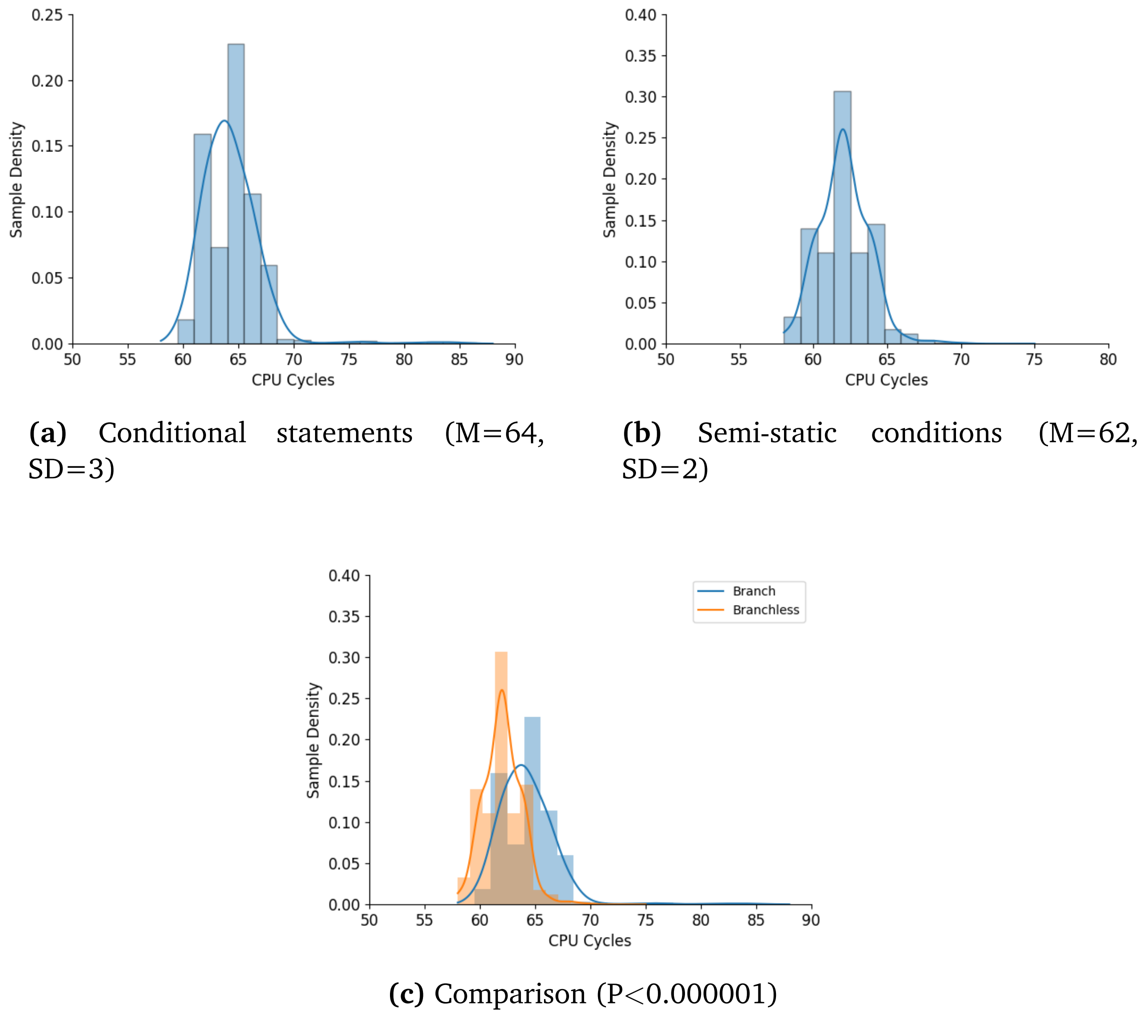
(a) Switch statements (M=30, SD=8)

(b) Semi-static conditions (M=8, SD=1)

(c) Comparison

Figure 4.9: CPU cycle measurements for a 5 case switch statement versus semi-static conditions with unpredictable conditions in the hot-path. Empty functions are used as branches for ease of testing.

statements, unpredictable conditions yield similar cycle distributions to the preceding examples which can be seen in Figure 4.9. Using the current methodology where random conditions are generated in the range of 0 to $n-1$ where n is the number of branches, as $n \to \infty$ the misprediction rate tends to 1 and hence distributions become uni-modal and skewed to higher cycle numbers. This is entirely expected; if conditions are random and constantly changing, then the probability of predicting the correct branch is inversely proportional to the number of branches, which tends to zero as the number of branches tends to infinity. Typically on GCC, large if-else chains or switch statements are optimised into jump tables which organise branches at distinct memory locations in a particular structure (binary tree for example), and utilise indirect jumps to computed offsets to facilitate branch taking. This additional indirection does incur more significant penalties when mispredicted owing to the additional data dependencies in the pipeline; extrapolating the median latencies from the predicted (M=13, SD=2) and mispredicted (M=31, SD=3) branches in Figure



(a) Conditional statements (M=64, SD=3)

(b) Semi-static conditions (M=62, SD=2)

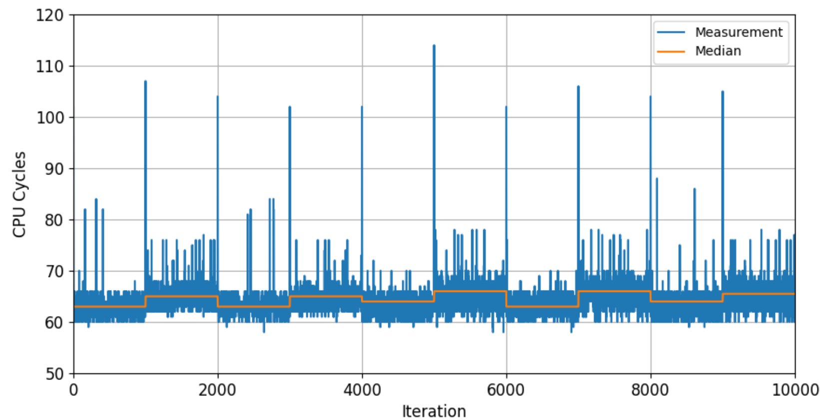
(c) Comparison (P<0.000001)

Figure 4.10: CPU cycle measurements for conditional branching versus semi-static conditions with predictable branching, conditions change every 1000 iterations.

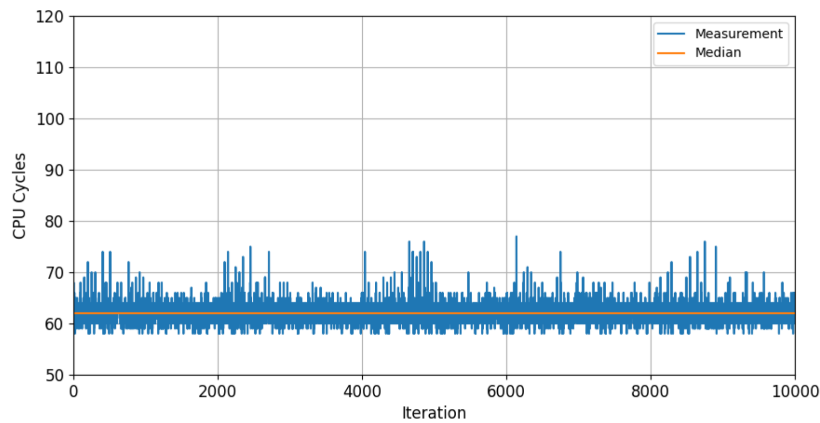
4.9(a), the misprediction cost is estimated to be 18 cycles in this example!

It is clear that for this particular use case, semi-static conditions offer superior performance to conditional branching when misprediction rates are high and branch-changing can be isolated in cold code paths. In the context of HFT, speed is paramount, and shaving off several nanoseconds per branch in the critical path offers an edge in order execution latency against competitors and can result in more profitable trading.

General use cases Whilst hot-path optimisation of mispredicted branches is not necessarily tied to HFT, but can be expanded to general performance critical sections of code in various applications (e.g., gaming, aerospace, infrastructure), an interesting investigation would be for general use cases where branches are not necessarily unpredictable (in terms of conditions). Earlier investigations revealed that branch-taking has comparable latency with isolated function calls, and in compar-



(a) Conditional statements



(b) Semi-static conditions

Figure 4.11: CPU cycle measurements per iteration for conditional branching and semi-static conditions.

ison to conditional statements, have marginally less assembly to execute to facilitate branch taking since there are no runtime checks. The following investigations outline instances where these subtle differences manifest in increased performance, using the same methodology of separating branch-changing from hot-path measurements. To simulate predictable branches, the same test suite used for specialised applications can be adapted to changing the boolean conditions based on a regular interval, rather than randomly generating them.

Figure 4.10 reveals that even when branches are predictable, semi-static conditions are slightly more performant in terms of branch-taking. In contrast with earlier tests, the latency of conditional statements form a uni-modal distribution since the misprediction rate is close to zero, nevertheless the extension of the trend-line to the 80-85 ns region shows there are mispredictions, but are rare. What is fascinating is that the core distribution of measurements for conditional branching is shifted to

higher latencies in comparison to semi-static conditions, when execution latencies for predicted branches should be the same. The key questions that arise are what is the cause of this shift, and how significant is the contribution of mispredictions to the overall median latency at different switching intervals. To answer the former, a subset of measurements for both semi-static and conditional branches can be extracted and plotted to visualise the execution latencies per iteration.

Figure 4.11(a) reveals some remarkable behaviour. Firstly, the misprediction penalty can be seen distinctly by sharp increases in latency at regular 1000 iteration intervals, this is indeed directly caused by changing the branch direction and seems to be corrected relatively quickly, which is indicative of an n-bit prediction scheme. Every time the branch direction changes, the median shifts by 2 or 3 cycles forming the observed saw-tooth pattern. Upon inspection of the underlying assembly, this behaviour is an artefact of the compiler reordering the conditional statement's assembly such that the backward branch (if) experiences slightly lower execution latencies than the forward branch (else) due to additional jumps around the code segment.

```
000000000000305c:    mov    rdi,QWORD PTR [rip+0x42e5]
0000000000003063:    shl    rdx,0x20
0000000000003067:    mov    rsi,rax
000000000000306a:    or     rsi,rdx
000000000000306d:    cmp    BYTE PTR [rip+0x42dc],0x0
0000000000003074:    je     3090 <_measure+0x40>
0000000000003076:    call   2c00 <send_order>
(...)
0000000000003090:    call   2c40 <_adjust_pricing>
0000000000003095:    jmp    307b <_measure+0x2b>
```

These kind of assembly ordering semantics have even more prevalent effects for switch statements (5-6 cycles faster!) which can be seen figure 4.9(c) by comparing the distributions of semi-static conditions and predicted branches for switch statements. This is mainly due to the additional assembly required perform the necessary computations when preparing to traverse jump tables:

```
00000000000041fd:    shl    rdx,0x20
0000000000004201:    mov    rbx,rax
0000000000004204:    or     rbx,rdx
0000000000004207:    cmp    QWORD PTR [rip+0x5161],0x4
000000000000420f:    ja     4235 <_measure+0x45>
0000000000004211:    mov    rax,QWORD PTR [rip+0x5158]
0000000000004218:    lea    rdx,[rip+0x1e05]
000000000000421f:    mov    rdi,QWORD PTR [rip+0x5142]
0000000000004226:    movsxd rax,DWORD PTR [rdx+rax*4]
000000000000422a:    add    rax,rdx
```

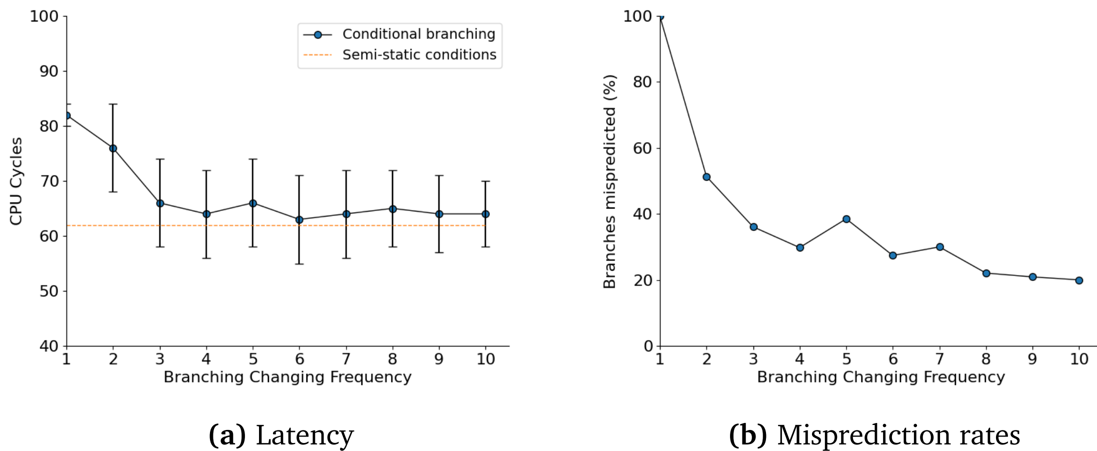
```
000000000000422d:        jmp     *rax
0000000000004230:        call    3ed0 <_send_order_1>
(...)
0000000000004249:        nop     DWORD PTR [rax+0x0]
0000000000004250:        call    3eb0 <_send_order_n>
0000000000004255:        jmp     4235 <_measurev+0x45>
0000000000004257:        nop     WORD PTR [rax+rax*1+0x0]
(...)
```

When comparing this to the underlying assembly generated for semi-static conditions, it becomes apparent that the absence of runtime checks is the sole contributor to the lower execution latencies observed.

```
000000000000309d:       mov     rdi,QWORD PTR [rip+0x42a4]
00000000000030a4:       mov     rbx,rax
00000000000030a7:       shl     rdx,0x20
00000000000030ab:       or      rbx,rdx
00000000000030ae:       call    2b00 <_branch_>
```

In the context of general usage for 'static branches' (conditions that change infrequently), the results have several implications. The power of semi-static conditions has always been the ability to change the direction of a branch programatically. At compile time, the programmer has little influence over such re-orderings, and although an informed prediction can be made about the more likely direction of the branch (forward or backward), if this where to change at runtime, the programmer will be paying at least 2-3 cycles per iteration which will start to add up. This allows for the `set_direction` method to be used more freely; rather than isolating it in the cold path, if the branch direction is changed relatively infrequently then the cost of code modification will amortise itself over many iterations of cheap branch taking.

In regards to contributions from branch mispredictions, the median latency and standard deviations for conditional branching where measured for varying branch changing frequencies (in terms of number of iterations passed before the condition is changed), along with the associated misprediction rates which can be shown in Figure 4.12. These results are quite unexpected; even for the branch changing every every iteration one would expect the BPU to spot this relatively simple pattern of taken not-taken, however this is not the case. Even when the conditions change at regular intervals in a predictable manner, the organisation of where these instructions lie within the executable effect the BP's ability to make predictions based on history correlations. In these benchmarks, the actual 'path' that is being measured resides in a function to help maintain assembly ordering for fair and precise measurements, which is called within a tight loop and performs a significant amount of computation per iteration. This can add noise, however the more likely reason is that the BP is unable to correlate the iteration count with the condition being evaluated (at regular intervals). The actual predictive mechanism can be deduced through the results; when branches are changed every iteration, misprediction rates



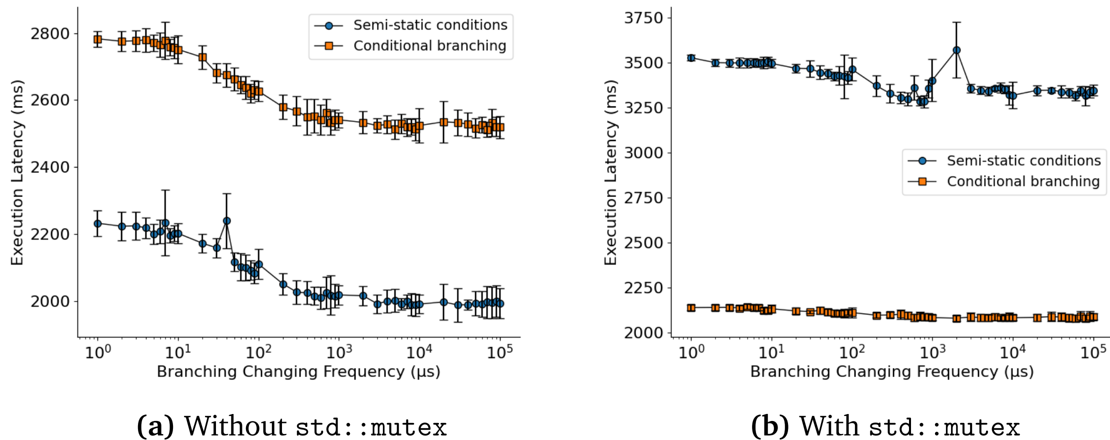
(a) Latency **(b)** Misprediction rates

Figure 4.12: CPU cycle measurements and respective misprediction rates for conditional branching at varying branch changing frequencies (number of iterations per condition change).

are close to 100% which results in significantly higher latencies but tighter standard distributions since all branches are mispredicted. As branch changing frequency decreases, the misprediction rate follows as well as the median latency, but standard deviations increase since measurement distributions contain a mix of predicted and mispredicted branches. From the data, it appears that there are 1.5-2 mispredictions per condition change, which is synonymous with a 2-bit saturated counter prediction scheme (hinting towards TAGE-like predictors used in modern Intel processors)! When misprediction rates are high, the associated penalties are the major contributor to increased latencies and standard deviations. When branches are better predicted and conditions change relatively infrequently, differences in latency are attributed to the underlying assembly generated by the compiler, resulting in more subtle performance differences.

So far general application benchmarks have been conducted on the cycle level, an interesting investigation is to see how they manifest in larger systems. The next test involves a multi-threaded benchmark where branch-directions are changed at regular time intervals, the branches perform a relatively simple computation and store the result in an array to prevent optimisation. Whilst the example is simplistic, it represents a system that polls events on a worker thread and changes flags that are evaluated in continuous loops, an example of such application is feature-flag selection for larger code-bases.

```
static void benchmark(benchmark::State& s)
{
    int results[2];
    std::thread worker(poll_events);
    for (auto _ : s)
    {
        for (int i = 0; i < iterations; i++)
```



(a) Without `std::mutex` **(b)** With `std::mutex`

Figure 4.13: Benchmark results for semi-static conditions versus conditional branching in multi-threaded application where branches change at regular time intervals. Mutex's have been applied to semi-static conditions exclusively. Each simulation was run for 10^8 iterations.

```
        {
            if (condition)
                results[flag] += action_1();
            else
                results[flag] += action_2();
        }
    }
    worker.join();
}
```

Even though the observed changes are relatively small, occasionally they can manifest as large performance gains which can be seen in Figure 4.13(a). When using the language construct in a multi-threaded environment, there may be a chance that the wrong branch is executed since the `set_direction` method is not atomic (discussed in more detail in Section 5.3). Using synchronisation will prevent this, however it results in large performance degradation which can be seen in Figure 4.13(b). Regardless, the proposed language construct shows immense promise in performance optimisation in general settings; any places where misprediction rates cause performance bottlenecks or branch-taking is slow due to surrounding code, semi-static conditions offer a convenient alternative to more efficient branch-taking.

4.5 Summary of Conclusions

The foregoing investigations have not only provided a transparent performance analysis of the various methods that comprise semi-static conditions, but have been successful identifying numerous use cases where mispredicted branches can be opti-

mised heavily with the use of the language construct.

By employing a number of sophisticated testing suites adapted from architectural forums and prior literature, section 4.3 effectively delved into the underlying schematics that drive the performance of branch-changing and branch-taking methods. Concerning branch-changing, the minimal isolated cost of assembly editing was evident; however, it was discovered that temporal locality of assembly editing, to when the edited instructions are executed, directly influenced the hardware penalties associated with SMC clears. The mechanics behind this phenomenon are well understood, and whilst SMC penalties where unavoidable due to architectural constraints, they where able to be isolated within `set_direction` exclusively. Leveraging preemptive conditional evaluation, the control over branch directions can be guided away from performance-sensitive code sections. As for branch-taking, which invariably exists within the critical path, performance was optimized to the theoretical limit. The only disparities observed where in additional instruction latencies when compared to conventional calls. While there is a slight initial penalty for branch-taking after assembly modification, it was discerned that this setback could be mitigated through BTB warming, a possibility not readily achievable in the same manner with conditional branch prediction.

Using these newfound insights, branch execution performance of semi-static conditions where compared against conditional statements in two distinct scenarios; when branches are poorly and well predicted. In the case of poorly predicted branches, semi-static conditions optimise branch taking through avoiding the runtime check, and hence avoid the associated misprediction penalties. The absence of runtime checks also improves performance for well predicted branches owing to the fewer assembly instructions that need to be executed, such effects where found to be more prevalent in large switch statements that are converted into jump tables. The former findings suggest that semi-static conditions have the potential to be used for branch-optimisation in larger low-latency systems.

Chapter 5

Evaluation

This section will evaluate the software contribution based on the prototype developed in Chapter 3, along with the experimental methods employed to accurately benchmark the efficiency of semi-static checks. Section 5.1 starts by evaluating the overall approach taken to developing semi-static checks in comparison to alternative viable solutions. Section 5.2 analyses the safety of the language construct through a combination of static and runtime analysis approaches, these methods are also employed in section 5.3 which focuses more on reliability in terms of behaviour and synchronisation. Section 5.4 touches on portability to different architectures and operating systems, and section 5.5 focuses on the experimental methods employed for efficiency benchmarks and outlines more appropriate tests for industry settings.

5.1 Overall Approach

The goal of semi-static checks is to provide programmers control over conditional branching; reaping the performance benefit of zero-runtime checks during branch-taking, whilst being able to change the direction of a branch at runtime. This kind of behaviour cannot be attained through conventional means available in the C++ standard; particular "branches" can be generated at compile-time based on conditions through template instantiation, but the ability to do this with runtime generated conditions is fundamentally impossible since templates are a compile-time phenomenon. Mechanisms for switching between function calls exist without explicit conditional branching in the form virtual inheritance facilitated by virtual-table look-ups and function pointer de-referencing. However in reality this is slower than conditional branching in most scenarios even when branches are often mispredicted, and in the case for virtual functions, a good optimiser will often speculatively de-virtualise calls into conditionals. At the hardware level, these types of indirect calls require not only predictions for conditions but also data in the form of memory addresses (for example, a virtual-table lookup takes roughly 3 memory accesses before the `call` address is resolved), and as a result suffer from higher misprediction penalties and are more vulnerable to adverse cache-related effects in oppose to regular conditional statements which have a smaller instruction cache footprint.

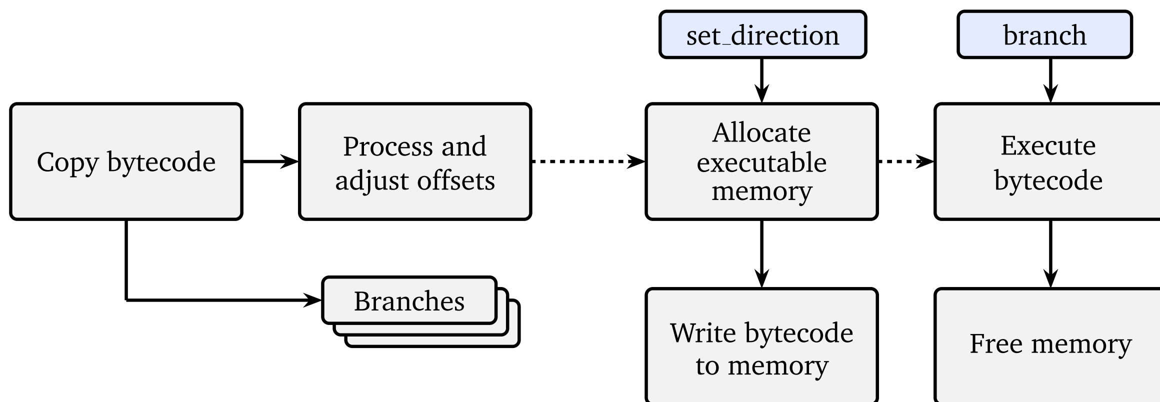


Figure 5.1: State diagram of semi-static conditions internals for both setup and execution using JIT-style runtime assembly generation.

This lack of flexibility meant that only assembly editing can facilitate the desired behaviour: fast deterministic branch-taking controlled by the programmer through a slower auxiliary interface. The reality of this is multifaceted. SMC is non-standard compliant and is widely considered a poor programming practice owing to complex maintenance, debugging and portability, despite it being used abundantly in debuggers and the Linux kernel. Given these concerns, the goal of development was to minimise the assembly editing component whilst maximising the simplicity of design for maintenance and portability reasons. However this is not the only viable approach.

Run-time code generation (JIT) An alternative approach to SMC facilitated branch-changing is a just-in-time (JIT) approach, which has become popular in modern compilers and interpreters. Whilst JIT is generally more widely accepted form of runtime assembly manipulation, there are some inherent issues that make it inferior to our approach. The first is complexity. Figure 5.1 shows the process of preparing the byte-code associated with the branches to actually executing them, with the majority of the complexity residing in the first two stages. Copying the byte-code is error prone: whilst it is simple to find the preliminary instructions for the function, being able to accurately determine when the function is ”finished” in memory requires architecture specific complex logic. For example, using a `ret` opcode as a proxy would not work as there can be multiple exit points, and differentiating an opcode from a byte offset would require accounting for instruction length which introduces a substantial amount of administrative overhead. Then comes the challenge of adjusting position dependant instructions within this byte-code to work, such as PC relative instructions, which further increases development complexity. The best way to do this would be to have an in house JIT compiler as part of the language construct, which is impractical.

Even if hypothetically the former challenges where addressed, the eminent problem that would prevent JIT from being used in low-latency settings is the means of which it can be executed in modern C++.

```
void* new_page = mmap(nullptr, 4096,
    PROT_READ | PROT_WRITE | PROT_EXEC,
    MAP_PRIVATE | MAP_ANONYMOUS,
);
std::memcpy(new_page, bytecode, sizeof(bytecode));
(...)
int (*add)(int, int) = (int (*)(int, int))new_page;
int result = add(5, 3);
```

Above is a sample of two necessary components required required to facilitate JIT-style assembly execution in C++: memory allocation and execution. From user space, the only way to execute this newly generated byte-code is through casting the address of the newly created executable page to a function pointer and de-fencing it, which is much more expensive at runtime than conditional branching. Whilst runtime code generation could offset SMC machine clears since instructions are not being modified and executed, rather just generated, the overhead of pointer de-referencing in latency critical paths defeats the whole purpose of the language construct.

Assembly editing In comparison to JIT, the method used to develop semi-static conditions has many advantages. The first is clearly simplicity as runtime assembly editing using an intermediary trampoline function (`branch` method) is facilitated through a 4-byte `memcpy` at runtime. The simplicity of the concept directly translates to simplicity in development which is reflected by the overall size of software artefact. In terms of maintainability, which is important for a library which employs architecture specific optimisations, a simple implementation will easier adapt to future changes. From a low-latency application perspective, less code translates to less assembly instructions which have smaller instruction cache footprints, a favourable property for systems that prioritise keeping as much latency-critical code in lower level caches. The second advantage is branch-taking speed; using the assembly editing approach the branch-taking overhead is simply the overhead of a relative jump, which has superior prediction schemes from pointer de-referencing in JIT. A downside of the current approach is the SMC penalties incurred during branch direction changing. Even in a hybrid approach where the trampoline can be hard-coded on separate executable pages, editing existing assembly is unavoidable. Whilst avoiding SMC penalties seems impossible using this scheme on modern architectures, it does open some interesting investigations in mitigation schemes, but nevertheless the trade-off for superior branch-taking ought to be favourable for these sorts of applications.

5.2 Safety

Assembly editing gives rise to undefined behaviour with respect to the C++ standard, and can bring forth security vulnerabilities. The first issue that can arise is

that one or more of the branches lie at a signed displacement greater than 2^{32} bytes from the `jmp` instruction within the `branch` method. In this instance the program will redirect control flow to an area in the code segment which does not belong to any of the branches, resulting in adverse behaviour. Instances like this ought to be caught out as early as possible. To simulate this, two arbitrary function pointers with a displacement greater than 2^{32} are created and passed into the `BranchChanger` constructor as such:

```
using ptr = int(*)(int, int);
ptr func_1;
ptr func_2 = reinterpret_cast<ptr>(
    reinterpret_cast<intptr_t>(func_1) +
    (static_cast<intptr_t>(1) << 34)
);
BranchChanger branch(func_1, func_2);
```

As a result the following runtime exception is raised upon instantiation:

> terminate called after throwing an instance of 'branch_changer_error'
>
> what(): Supplied branch targets (as function pointers) exceed a 4GiB displacement from the entry point in the text segment, and cannot be reached with a 32-bit relative jump. Consider moving the entry point to different areas in the text segment by altering hot/cold attributes.
>
> Aborted (core dumped)

Another form of undefined behaviour stems from more than one instance of semi-static conditions being present at any given time during program execution. In this scenario, the conflicting instances of `BranchChanger` will share the same `branch` entry point due to template specialisation and compete for assembly editing of a single `jmp`. As a result, branches that do not belong to the immediate `BranchChanger` instance may be executed. When multiple instances do exist, the following error is raised:

> terminate called after throwing an instance of 'branch_changer_error'
>
> what(): More than once instance for template specialised semi-static conditions detected. Program terminated as multiple instances sharing the same entry point is dangerous and results in undefined behaviour (multiple instances write to same function).

```
Aborted (core dumped)
```

The former exception handling measures are effective at detecting the main behaviours associated with assembly editing that can result in detrimental effects later on in program execution. This, along with other higher level behaviour exceptions, have been integrated into a rigorous automated testing suite part of the library, allowing programmers to test the build prior to usage for peace of mind.

In the context of security, the primary vulnerability that arises using semi-static conditions is through changing executable page permissions to read/write/execute. Often times during program execution, multiple pages and hence multiple address ranges are vulnerable to memory modification from other processes, allowing attackers to inject their own code, access sensitive data or tamper with sensitive information. From a mitigation standpoint, eliminating this vulnerability completely is not possible, since page permissions need to be altered at least temporarily to facilitate assembly editing. What can be done is to minimise the transient window of vulnerable address ranges by performing page permission modification and reversion exclusively within the `set_direction` method, which can be activated by the programmer through compiler flags for example.

```
void set_direction(bool condition)
{
    if (condition != direction)
    {
        change_page_perimissions(bytecode,
            PROT_READ | PROT_WRITE | PROT_EXEC
        );
        std::memcpy(src, dest[condition], DWORD);
        change_page_perimissions(bytecode,
            PROT_READ | PROT_EXEC
        );
        direction = condition;
    }
}
```

Using this method, programmers can be ensured that the risk of such exploits are minimised. To ensure that *safe-mode* is respected throughout the program lifetime, programmers can add the following flag upon compilation:

```
-DSAFE_MODE
```

However this comes with a cost. In a low-latency setting, using the secure branch-changing method introduces higher execution times and larger standard deviations owing to the two system calls used to alter page permissions, in addition to increased

pressure on instruction-caches. When thinking of general cases as outlined in Section 4.4, the more expensive cost of branch-changing has negative implications on amortisation, and will likely limit the use in scenarios where branches are well predicted. This is the inherent trade off in low-latency settings; security versus speed. In commercial software that is exposed directly to the user, such security ought to be included, however in the case of secretive proprietary software (such as trading systems) such security measures may not be necessary. In light of this, both API's are exposed to the programmer, whom can make an informed decision what to include.

5.3 Reliability

In terms of reliability, the main areas of focus in the context of semi-static conditions are correctness and consistency of behaviour. From a high-level perspective, an exhaustive formal proof showing that semi-static conditions have equivalent behaviour to conditional statements is unnecessary. In the context of semi-static conditions, static analysis of program behaviour does not bear much meaning since the program state itself is not static due to self-modifying assembly instructions! To evaluate correctness a test suite can be set up by running a tight loop that (1) changes the branch direction and (2) takes the branch successively and see if the wrong branch is taken based on the runtime condition.

```
while (run)
{
    branch.set_direction(condition);
    branch.branch(counter);
    condition = !condition;
}
```

Under these conditions, in a single threaded environment semi-static conditions will **always** exhibit correct behaviour, however this may not always the case in multi-threaded setting. In the development process it was mentioned that the `branch` method is typically aligned on 16-byte boundaries and is sufficiently small to fit entirely on a single cache line. Under the C++ standard, there are no guarantees that stores to executable memory are thread safe (though this is not explicitly stated). However at the architectural level, x86 guarantees writes to a single cache line to be atomic [68]. Given the jump instruction being patched never crosses multiple cache lines due to its size and the alignment of the parent sub-procedure, a single 4 byte `mov` operation followed by SMC clears is sufficient for ensuring synchronisation across threads on a single core. Even if modification occurs from a different core, cache coherency protocols will ensure consistency, meaning subsequent fetches (which are atomic for instructions on 16 byte boundaries) will reflect the modified code [68]. Nevertheless assembly modification using semi-static conditions manifests as the `set_direction` method which is not an atomic operation, but rather an entire sub-procedure, so in theory the wrong branch could be executed solely due

to instruction interleaving. However such effects where unable to be observed experimentally, and likely to be extremely rare and dependant upon the underlying scheduling algorithms employed by the OS.

Although the overall focus in multi-threaded environments where limited, it would be interesting to observe if serialising instructions such as `lfence` and `mfence` can be used as a synchronisation mechanism unilaterally without compromising the performance of branch-taking in future investigations.

5.4 Usage and Portability

The software artefact is packaged as a static library which can be incorporated into project using the CMake build system. The decision to package semi-static checks as a static library over a dynamic library is due to performance reasons; static libraries present all code in the executable at compile time whereas dynamic libraries need to be loaded by the OS at runtime. Using the dynamic library methods require an extra layer of indirection through symbol table look-ups, and parts of the library themselves are brought into memory on-demand which is prone to cache misses and page faults, all of which contribute to jitter which is undesirable in low-latency setting. The drawbacks of using static libraries are increased compilation times and the size of the executable, which is made more prominent given the template-heavy implementation of semi-static conditions.

Cross platform builds on C++ are notoriously messy. CMake abstracts the differences between various build systems and compilers on different platforms which simplifies development and usage greatly. From a portability standpoint, the CMake pre-processor can be configured to selectively include architectural specific headers into the final build which is extremely beneficial given the low-level nature of the library. To begin the build (assuming the library is cloned into the same directory), users simply first specify a build directory as such:

$ cmake -E make_directory ”build”

Next, the build system files can be generated in the newly created directory with

$ cmake -E chdir ”build” cmake ../

Then finally built with

$ cmake –build ”build”

Subsequent tests can be run to validate the build system using

Compiler	Windows		MAC		Linux	
	x86-64	ARM64	x86-64	ARM64	x86-64	ARM64
GCC			✗	✗	✓	✓
MSVC	✓	✓	✗	✗		
Clang			✗	✗	✓	✓

Table 5.1: Compatibility matrix for the semi-static conditions library. Ticks are given to platforms where the library has been tested and is functional, and crosses are given to untested platforms and/or no functionality. Empty cells represent when the platform combination does not represent a native build.

```
$ cmake -E chdir "build" ctest
```

To use the library, the generated archive must be compiled and linked against the branch library (libbranch.a) as such

```
$ g++ mycode.cpp -std=c++17 -isystem semi-static-conditions/include -Lsemi-static-conditions/build -lbranch -o mycode
```

This necessitates the entire setup required to use the library, the only requirements being a relatively new version of CMake (version 3.2 and above) to facilitate the building! The core `BranchChanger` construct can be used by including the following header:

```
#include <branch.hpp>
```

Portability was a challenge from a development perspective. The implementation of semi-static checks utilises OS specific system calls, compiler specific attributes, in-line assembly, and architecture specific offset computations which can all manifest in different combinations depending on the users system configuration. In terms of achieving the correct build, a dedicated single header file was utilized to define the platform using a series of pre-processor directives. These directives are employed throughout the library to enable the incorporation of architecture-specific, compiler-specific, and OS-specific code. It is possible to delegate this process entirely to CMake, however such an approach would necessitate users to specify the platform using a series of compiler flags which is rather tedious, and therefore avoided.

For this initial version, popular compilers and operating systems where targeted for compatibility, on both x86-64 and ARM64 architectures. Builds where tested by invoking the library using the CMake steps shown above on different systems, results of the tests (including if the library was unable to tested) are shown in Table 5.1. Unfortunately, semi-static conditions is not portable on Apple silicon owing to the Hardened Runtime OS security feature which prevents page permissions from being

changed to write/execute. Some of these security features can be disabled allowing for binary editing of runtime-allocated pages using the JIT approach in Section 5.1, however as discussed, does not meet the latency requirements for this construct.

5.5 Experimental Method

A great focus of this report was to evaluate the performance of semi-static conditions against the current state of the art and rationalise any observed behaviours and inefficiencies from a micro-architectural perspective. Due to the sensitivity of the measurements, careful consideration was taken in developing a measuring suite that prioritises the accuracy and reproducibility of results. The choice of measuring instruments where sensible; reference cycle counters where sufficient in producing accurate results allowing for the observation of subtle behaviours on the hardware level. However a number of improvements can be made to the overall production setup in the context of OS tuning and networking.

Given the primary objective of this report which centers around exploring innovative branch optimization techniques for HFT, it becomes imperative that the OS chosen for conducting performance benchmarks mirrors an industrial setup to the greatest extent possible. The selection process for the system was driven by a preference for a Linux-based High-Performance Computing (HPC) environment featuring a Intel i7 processor. Additionally, a cloud-based Virtual Machine (VM) with an Intel Xeon chip was utilized. It is, however, important to note that in both cases, complete root privileges were not attainable. This limitation had significant implications, particularly in the domain of kernel tuning. Specific adjustments related to CPU scaling and scheduling could not be replicated to the same extent as they would be in a genuine HFT system. This discrepancy is noticeable in the variations and distributions observed in the recorded measurements. To address this challenge, a nuanced approach was adopted. The measurement suite was meticulously tailored to execute numerous iterations, allowing the CPU to stabilize at an optimal operating frequency. This strategic approach effectively minimized measurement variance, enabling subsequent interpretation through statistical tests. While it remains true that with meticulous kernel tuning, the necessity for extensive statistical treatment could be substantially diminished, leading to clearer and more refined data, the outcomes presented in the report successfully fulfilled the intended communication goals. Despite the existing constraints, the data provided within the report aptly conveyed the essential insights and findings.

The second improvement is of a more comprehensive nature, and its attribution extends to the entire system. An inherent limitation associated with microbenchmarking in isolation, particularly when examining minute differences at the low-nanosecond scale, is that the tests themselves possess an artificial nature, deviating from the essence of a genuine production environment. Despite the endeavors made in Section 4.4 to craft test suites that emulate a pseudo-realistic production environment in terms of computational workload, the actual representation of behavioral changes remains somewhat imprecise when contrasted with a fully functional HFT system. Yet, it's important to acknowledge the inherent limitations in addressing

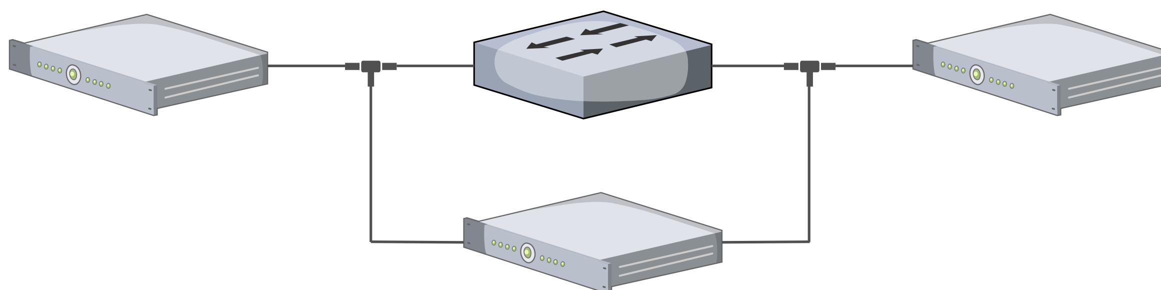


Figure 5.2: Ideal productionised setup of benchmarking suite for HFT applications. Left server replays market data across a high speed ethernet cable, the switch in the middle is equipped with high precision time-stamps, server on the right is the production system to be measured, and server in the middle computes response times. Adapted from [5].

this issue. The proprietary nature of trading firms code renders it a closely guarded trade secret, largely due to its direct influence on profitability. The epitome of an ideal experimental arrangement is presented in Figure 5.2, a concept initially proposed by Carl Cook as the most robust methodology for micro-benchmarking latency enhancements within a trading system [5]. It's worth noting, however, that such a setup warrants its own comprehensive report and entails substantial complexity and associated costs.

Chapter 6

Ethics

The development of software that is intended to be used in low-latency environments to facilitate faster decision making has the propensity for adverse dual-usage. This project has the potential for usage in military applications, particularly technologies that require fast-decision making or response times, for example target tracking and navigation systems. Usage in military applications should be in accordance with domestic and international law, and should be evaluated and regulated accordingly.

The focus of developing novel low-latency optimisations for HFT raises its own concerns. Whilst HFT provides liquidity and makes markets more efficient, the same trading systems can be used for manipulation through order book spoofing and layering without following through on trades, creating an illusion of supply and demand. In addition, HFT firms operate on superior technology and information advantages to retail investors making competition on the high-frequency scale virtually impossible. Addressing these ethical concerns requires a combination of regulatory oversight, industry self-regulation, transparency initiatives, and public awareness. Striking a balance between technological innovation and ethical responsibility is essential for ensuring the integrity and stability of financial markets in the era of HFT.

Chapter 7

Conclusions

This report has shown that software-level branch optimisation can be achieved using a novel language construct, semi-static conditions, offering superior execution latencies to the current state of the art in both specialised and generalised scenarios. There is no doubt that the methodologies applied to facilitate this optimisation are unconventional. The use of assembly editing has long been shunned from a software development and micro-architectural perspective, however it seems that revisiting this old yet interesting nuance of low-level development has opened new doors for low-latency optimisation.

The notion of semi-static checks is simple, remove runtime checks by separating branch-taking from condition evaluation, however as we have seen throughout the report this is quite a surface level interpretation. What really is happening is a trade of branch-prediction schemes on the hardware level which is facilitated through assembly editing. In semi-static checks, the conditional statement is re-engineered as polymorphic relative jump instruction which can be controlled directly by the programmer through an auxiliary interface. As an artefact of this modification, the idea of execute-stage branch mispredictions are completely eliminated. The result? A powerful decoupling that optimises branch-taking to the theoretical limit. Results in Sections 4.3 and 4.4 showcased the superior performance of semi-static conditions against conditional branching at high misprediction rates, saving anywhere from 6-18 cycles on average, a direct artefact of removing runtime conditional checks from branch-taking. Even when branches are well predicted, semi-static conditions exhibit faster branch-taking as a consequence of a cleaner control path and fewer instructions on the machine code level, saving 2-6 cycles per execution, allowing the expensive branch-changing method to be amortised in single and multi-threaded scenarios.

The phenomenon of semi-static checks brings forth many avenues for further investigation, particularly around the application of self-modifying binaries for program optimisation, and minimising any adverse hardware effects. Section 4.3 took a detailed look at the behaviour of writing instructions into executable memory and revealed that locality between assembly editing and the assembly being executed results in severe processor penalties in the form of SMC machine clears. Though it is understood that locality in terms of caching, prefetching and paging has a proportional effect on the severity of SMC penalties, this investigation was unable to

properly quantify the effect. Further investigations into this would allow the branch-changing portion of the construct to be further optimised, in addition to providing the foundational research necessary to fully understand the effects of SMC on the micro-architectural level. In addition, the general idea of making targeted and granular changes in running executables to optimise various language level intrinsics should be investigated more broadly. Perhaps the general idea of substituting more BPU-friendly control flow instructions and modifying them can be applied to many different types of branches, more specifically dynamic dispatch and any form of register jumps.

In sum, the absence of language level-optimisations for hardware based branch prediction has birthed a new toolkit in tackling optimisation problems in low-latency setting. With this report pioneering the use of assembly editing for branch optimisation, we hope it serves as a benchmark for future investigations that apply this long-neglected and interesting programming practice to more complex problems surrounding optimisation.

Bibliography

[1] Lin CK, Tarsa SJ. Branch prediction is not a solved problem: Measurements, opportunities, and future directions. arXiv preprint arXiv:190608170. 2019. pages 1, 8

[2] Aldridge I. High-frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems. Wiley trading series. John Wiley & Sons, Incorporated; 2013. Available from: https://books.google.co.uk/books?id=kHPwjgEACAAJ. pages 2, 13

[3] Cartea Á, Jaimungal S, Penalva J. Algorithmic and High-Frequency Trading. Cambridge University Press; 2015. Available from: https://books.google.co.uk/books?id=5dMmCgAAQBAJ. pages 2

[4] Core C++ 2019 :: Nimrod Sapir :: High Frequency Trading and Ultra Low Latency development techniques; 2019. Available from: https://www.youtube.com/watch?v=_0aU8S-hFQI. pages 2, 13

[5] CppCon 2017: Carl Cook “When a Microsecond Is an Eternity: High Performance Trading Systems in C++; 2017. Available from: https://www.youtube.com/watch?v=NH1Tta7purM. pages 2, 11, 13, 66

[6] CppCon 2014: Chandler Carruth ”Efficiency with Algorithms, Performance with Data Structures”; 2014. Available from: https://www.youtube.com/watch?v=fHNmRkzxHWs. pages 2

[7] Trading at light speed: designing low latency systems in C++ - David Gross - Meeting C++ 2022; 2022. Available from: https://www.youtube.com/watch?v=8uAW5FQtcvE&t=2875s. pages 2

[8] Branchless Programming in C++ - Fedor Pikus - CppCon 2021; 2021. Available from: https://www.youtube.com/watch?v=g-WPhYREFjk&t=1723s. pages 2

[9] Stroustrup B. An overview of C++. In: Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming; 1986. p. 7-18. pages 2, 11

[10] Stroustrup B. The design and evolution of C++. Pearson Education India; 1994. pages 2, 11

[11] Hunt G, Brubacher D. Detours: Binaryinterception ofwin 3 2 functions. In: 3rd usenix windows nt symposium; 1999. . pages 2

[12] Giffin JT, Christodorescu M, Kruger L. Strengthening software self-checksumming via self-modifying code. In: 21st Annual Computer Security Applications Conference (ACSAC'05); 2005. p. 10 pp.-32. pages 2

[13] GDB: The GNU Project Debugger; 2023. Available from: `https://www.sourceware.org/gdb/`. pages 2

[14] Eyerman S, Smith JE, Eeckhout L. Characterizing the branch misprediction penalty. In: 2006 IEEE International Symposium on Performance Analysis of Systems and Software; 2006. p. 48-58. pages 2

[15] Making Your Code Faster by Taming Branches; 2023. Available from: `https://www.infoq.com/articles/making-code-faster-taming-branches/`. pages 2, 13

[16] Bilokon PA, Lucuta M, Shermer E. Semi-static Conditions in Low-latency C++ for High Frequency Trading: Better than Branch Prediction Hints; 2023. Available from: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4553439`. pages 3

[17] Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Elsevier; 2017. pages 4, 5, 6, 7, 8, 22

[18] Noyce R, Hoff M. A History of Microprocessor Development at Intel. IEEE Micro. 1981 jan;1(01):8-21. pages 4

[19] S TANENBAUM A, Bos H. Modern operating systems; 2015. pages 4

[20] Moore GE, et al.. Cramming more components onto integrated circuits. McGraw-Hill New York; 1965. pages 4

[21] Stallings W. Computer organization. Architecture,:, Designing for performance. 2017;10. pages 4, 5

[22] McFarling S, Hennesey J. Reducing the cost of branches. ACM SIGARCH Computer Architecture News. 1986;14(2):396-403. pages 5, 6

[23] Smith JE, Sohi GS. The microarchitecture of superscalar processors. Proceedings of the IEEE. 1995;83(12):1609-24. pages 5

[24] Deitrich BL, Chen BC, Hwu W. Improving static branch prediction in a compiler. In: Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192). IEEE; 1998. p. 214-21. pages 5

[25] Emma. Characterization of branch and data dependencies in programs for evaluating pipeline performance. IEEE Transactions on Computers. 1987;100(7):859-75. pages 5, 7

[26] Flynn MJ. Computer Architecture Pipelined and Parallel Processor Design, 1995. Jones and Bartiett Publishers. pages 6

[27] Pan ST, So K, Rahmeh JT. Improving the accuracy of dynamic branch prediction using branch correlation. In: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems; 1992. p. 76-84. pages 7

[28] Fisher JA, Freudenberger SM. Predicting conditional branch directions from previous runs of a program. ACM SIGPLAN Notices. 1992;27(9):85-95. pages 7

[29] Mittal S. A survey of techniques for dynamic branch prediction. Concurrency and Computation: Practice and Experience. 2019;31(1):e4666. pages 7, 8, 9

[30] Smith JE. A study of branch prediction strategies. In: 25 years of the international symposia on Computer architecture (selected papers); 1998. p. 202-15. pages 7

[31] Yeh TY, Patt YN. Two-level adaptive training branch prediction. In: Proceedings of the 24th annual international symposium on Microarchitecture; 1991. p. 51-61. pages 8

[32] Yeh TY, Patt YN. Alternative Implementations of Two-Level Adaptive Branch Prediction. 1992;20(2). Available from: https://doi.org/10.1145/146628.139709. pages 8

[33] Seznec A. TAGE-SC-L branch predictors again. In: 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5); 2016. . pages 8, 9

[34] Zangeneh S, Pruett S, Lym S, Patt YN. Branchnet: A convolutional neural network to predict hard-to-predict branches. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE; 2020. p. 118-30. pages 8, 9, 10

[35] Sadooghi-Alvandi M, Aasaraai K, Moshovos A. Toward virtualizing branch direction prediction. In: 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE; 2012. p. 455-60. pages 9

[36] Seznec A. A 256 kbits l-tage branch predictor. Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2). 2007;9:1-6. pages 9

[37] Ozturk C, Sendag R. An analysis of hard to predict branches. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE; 2010. p. 213-22. pages 8

[38] Jimenez DA, Lin C. Dynamic branch prediction with perceptrons. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture; 2001. p. 197-206. pages 9

[39] Burcea I, Somogyi S, Moshovos A, Falsafi B. Predictor virtualization. ACM SIGOPS Operating Systems Review. 2008;42(2):157-67. pages 10

[40] Donadio S, Ghosh S, Rossier R. Developing High-Frequency Trading Systems: Learn how to implement high-frequency trading from scratch with C++ or Java basics. Packt Publishing; 2022. Available from: `https://books.google.co.uk/books?id=HBp2EAAAQBAJ`. pages 11

[41] Veldhuizen TL. C++ templates are turing complete. Available at citeseer ist psu edu/581150 html. 2003. pages 11

[42] Vandevoorde D, Josuttis NM. C++ Templates: The Complete Guide, Portable Documents. Addison-Wesley Professional; 2002. pages 11

[43] Abrahams D, Gurtovoy A. C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond. Pearson Education; 2004. pages 11

[44] Clang Language Extensions; 2023. Available from: `https://clang.llvm.org/docs/LanguageExtensions.html`. pages 11

[45] Other Built-in Functions Provided by GCC; 2023. Available from: `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`. pages 11

[46] Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering. 2012;2. pages 11, 12

[47] Smith AJ. Cache Memories. ACM Comput Surv. 1982 sep;14(3):473–530. Available from: `https://doi.org/10.1145/356887.356892`. pages 11

[48] Improving Performance by Better Code Locality; 2023. Available from: `https://easyperf.net/blog/2018/07/09/Improving-performance-by-better-code-locality`. pages 12

[49] Memory Part 5: What Programmers Can Do; 2023. Available from: `https://lwn.net/Articles/255364/`. pages 12

[50] C++ attribute: likely, unlikely; 2023. Available from: `https://en.cppreference.com/w/cpp/language/attributes/likely`. pages 12

[51] Menkveld AJ. High frequency trading and the new market makers. Journal of Financial Markets. 2013;16(4):712-40. High-Frequency Trading. Available from: `https://www.sciencedirect.com/science/article/pii/S1386418113000281`. pages 13

[52] The race to zero: Speech held at the International Economic Association Sixteenth World Congress. Bank of England; 2011. . pages 13

[53] MacKenzie D. A sociology of algorithms: High-frequency trading and the shaping of markets. Preprint School of Social and Political Science, University of Edinburgh. 2014. pages 13

[54] Arnoldi J. Computer algorithms, market manipulation and the institutionalization of high frequency trading. Theory, Culture & Society. 2016;33(1):29-52. pages 13

[55] Hasbrouck J, Saar G. Low-latency trading. Journal of Financial Markets. 2013;16(4):646-79. High-Frequency Trading. Available from: `https://www.sciencedirect.com/science/article/pii/S1386418113000165`. pages 13

[56] Zhang F. High-frequency trading, stock volatility, and price discovery. Available at SSRN 1691679. 2010. pages 13

[57] Brogaard J, Hendershott T, Riordan R. High frequency trading and the 2008 short-sale ban. Journal of Financial Economics. 2017;124(1):22-42. pages 13

[58] Brogaard J, et al. High frequency trading and its impact on market quality. Northwestern University Kellogg School of Management Working Paper. 2010;66. pages 13

[59] Boutros A, Grady B, Abbas M, Chow P. Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis. In: 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig); 2017. p. 1-6. pages 14

[60] Leber C, Geib B, Litz H. High frequency trading acceleration using FPGAs. In: 2011 21st International Conference on Field Programmable Logic and Applications. IEEE; 2011. p. 317-22. pages 14

[61] Fog A. Calling conventions for different C++ compilers and operating systems. Technical University of Denmark; 2023. pages 19, 27

[62] Knoop J, Rüthing O, Steffen B. Partial dead code elimination. ACM Sigplan Notices. 1994;29(6):147-58. pages 20

[63] Aga MT, Austin T. Smokestack: thwarting DOP attacks with runtime stack layout randomization. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE; 2019. p. 26-36. pages 21

[64] Corbet J. Kernel Development - Multi-protection VMA's. 2006. Available from: `https://lwn.net/Articles/182495/`. pages 21

[65] Corp I. Method for pipeline processing of instructions by controlling access to a reorder buffer using a register file outside the reorder buffer. 1996. Available from: `https://patents.google.com/patent/US5721855A/en`. pages 23

[66] Callahan D, Cooper KD, Kennedy K, Torczon L. Interprocedural constant propagation. ACM SIGPLAN Notices. 1986;21(7):152-61. pages 27

[67] Intel. A Technical Look at Intel's Control-flow Enforcement Technology. 2020. Available from: `https://www.intel.com/content/www/us/en/developer/articles/technical/`

technical-look-control-flow-enforcement-technology.html. pages 28

[68] Fog A. Test programs for measuring clock cycles and performance monitoring. Technical University of Denmark; 2023. pages 33, 39, 47, 62

[69] Godbolt M. Microarchitecture; 2016. Available from: https://xania.org/Microarchitecture-archive. pages 33

[70] Gabriele Paoloni IC. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures; 2010. Available from: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. pages 33, 34

[71] Kerrisk M. perf_event_open(2) — Linux manual page. 2023. Available from: https://man7.org/linux/man-pages/man2/perf_event_open.2.html. pages 35

[72] Google. Google Benchmark. 2023. Available from: https://github.com/google/benchmark. pages 36

[73] Corporation I. Intel® 64 and IA-32 Architectures Optimization Reference Manual; 2012. Available from: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. pages 38, 40

[74] Corporation I. Method and apparatus for self modifying code detection using a translation lookaside buffer; 1999. Available from: https://patents.google.com/patent/US6594734. pages 38

[75] Ragab H, Barberis E, Bos H, Giuffrida C. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In: 30th USENIX Security Symposium (USENIX Security 21); 2021. p. 1451-68. pages 39

[76] Fog A. Instruction tables. Technical University of Denmark; 2023. Available from: https://www.agner.org/optimize/instruction_tables.pdf. pages 42, 44