TU WIEN Informatics

Datalynx

An in-hardware framework for assessment of program similarity

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Fabian Philipp Posch
Registration Number 01456625

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Stefan Tauner

Vienna, 24th March, 2022

Fabian Philipp Posch	Andreas Steininger

Erklärung zur Verfassung der Arbeit

Fabian Philipp Posch

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. März 2022

Fabian Philipp Posch

Acknowledgements

My parents, whose patience I have been able to stretch generously as I worked on this very lengthy project. As well as my supervisor Stefan Tauner for removing roadblocks and sharing opinions about FPGA toolchains.

Lieven Eeckhout, Ajay Joshi, and Aashish Phansalkar, whose names I kept running into while digging through mountains of papers trying to understand what features of a program actually influence similarity. They have built much of the groundwork this thesis stands on.

I would also like to thank my good friend Charlie Long for providing the name *Datalynx* through a misread on my end.
As well as him, Taylor Wilkins, and Ashley Joy for helping me English good.

Finally I would like to thank F.D.C. Willard for his insights into related topics and for inspiring me to broaden my horizons.

Kurzfassung

Die Auswahl des korrekten Benchmarks zur Ermittlung der Eignung für den ausgewählten Anwendungsbereich oder für die anwendungsspezifische Hardwareentwicklung ist von essentieller Bedeutung. Die Messung von Ähnlichkeit von Programmen im Bezug auf Leistungscharakteristiken kann bei der Auswahl des richtigen Benchmarks zur Optimierung in einem bestimmten Anwendungsbereich helfen.
Wir präsentieren Datalynx, ein Hardware-basiertes Framework zur Messung von Programmvergleichbarkeit, welches die nötigen Werkzeuge direkt in Hardware zur Verfügung stellt. Wir besprechen zuerst vorangegangene Lösungswege zur Messung von Programmvergleichbarkeit und definieren daraus eine Liste von gemessenen Features für diese Implementation. Datalynx wurde als Zusatz für den open-source RISC-V-Kern RI5CY implementiert und im Pulpissimo-Mikrocontroller auf der Xilinx Zynq-7000-Plattform getestet. Diese Implementation erlaubt uns, Telemetrie zur Laufzeit zu ermitteln, wodurch zeitaufwendige Simulation als auch Einschränkungen der messbaren Daten sowie Auswirkungen auf die Leistungsfähigkeit im Vergleich zu einer Messung auf unmodifizierter Hardware wegfallen. Die Kernmodifikation *Enlynx* zeigt nur leichte Auswirkungen auf die Leistungsaufnahme des Kerns, es handelt sich jedoch um einen simplifizierten Machbarkeitsnachweis. Wir waren mit dieser Implementation in der Lage, die Funktionsfähigkeit einer solchen Lösung anhand von als Industriestandard gehandelten Benchmarks zu demonstrieren und die getesteten Benchmarks in drei Gruppen anhand ihres zeitlichen Verhaltens einzuteilen.

Abstract

Choosing the correct benchmarks for evaluating performance of a target task or task-specific hardware development is crucial. Measuring program similarity can help in selecting the right benchmark to optimize for the selected target.
We present Datalynx, a hardware-based framework built for measuring workload similarity, providing the tools necessary for recording data directly in hardware. We first discuss and compare previous approaches to measuring similarity of programs and benchmarks as well as define the set of features observed in this implementation. Datalynx was implemented as an add-on to the open-source RISC-V implementation RI5CY and tested within the Pulpissimo microcontroller on the Xilinx Zynq-7000 platform. This implementation allows us to gather telemetry in flight in realtime, eliminating the need for time-consuming simulations as well as overcoming limited availability of data access or performance impacts when performing these measurements on unmodified hardware. While being a proof of concept, the actual core modifications named *Enlynx* show minor hardware impact. We were able to demonstrate a functioning implementation of this approach as well as gather performance profiles from several industry standard benchmarks and categorize them into three main behavioral groups.

Contents

CHAPTER 1

Introduction

According to Wikipedia, a benchmark in computing is

> the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. The term benchmark is also commonly utilized for the purposes of elaborately designed benchmarking programs themselves. [Ben21]

We will use the word *benchmark* to refer to the latter, a (set of) program(s) constructed or assembled to gauge the performance of an artifact $\mathcal{A}$, in order to gain insight into how said artifact $\mathcal{A}$ might perform in a real world environment compared to a different artifact $\mathcal{B}$. However, even though tempting, using one simple performance score measuring some metric like execution time of an arbitrary program μ on artifact $\mathcal{A}$ will, due to the inherent complexity of any modern hardware design, always lead to information loss and thus to an inability of calculating accurate performance numbers for a yet unknown different arbitrary program ϕ. Providing the first steps to solving this issue will be the main focus of this thesis.

1.1 Motivation

Since the start of the new century, evaluating the quality of benchmarks has received rising scientific attention [EVB03, DD98, VB04, PJEJ05, EV03], especially for platforms supporting Linux-like operating systems. These papers mainly focus on reducing the amount of actual benchmarking programs in a suit, in order to speed up the process of evaluating a single system while retaining almost all information contained in the result.

However, to our knowledge, there exists little research on similarity of programs actually used in the field and benchmarks proposed for these micro processors.

When engineering a new core implementing a certain instruction set architecture (ISA), or building a product around such a core, it is imperative to be able to predict the real world performance in actual deployment. The lack of such knowledge can lead to a too powerful core being chosen in the best case, which adds cost and complexity, or a too weak core which, in the worst case, can lead to system failure. Knowledge about the adequacy of the chosen tools is thus essential. We take the first step in this exploration by providing the underlying tools necessary for measuring this information.

When working with microcontroller units (MCUs), only so much information can be extracted by simulation, as doing so is extremely slow [EVB03, KHH07] and some inputs cannot be easily simulated altogether, especially in real-time applications [KHH07]. It is possible to augment an existing binary with instructions meant to monitor performance counters [EV03], this however will inevitably alter the performance of the core on which the modified binary is run on as well as alter the frequency of hardware based events like cache misses if applicable. Obviously, our presented minimal FPGA-based implementation is far from an actual high performance core; such an implementation is left for further works. But it is easy to see that a hardware based solution monitoring an otherwise mostly untouched core is as close to real-world as possible.

1.2 Benchmark representativness

As mentioned, when selecting hardware platforms for a given application, knowing which platform is actually capable of handling the given task is crucial. For this reason, we employ benchmarks to measure certain characteristics of a given hardware artifact. This however requires the assumption of the final workload being similar enough to the used benchmarks. We say the benchmarks need to be *representative* of the final workload. There have been several attempts at characterizing the similarity of programs in the past. Most of the approaches presented here follow either a feature-agnostic or feature-aware methodology. In order to make this categorization, we use a slightly modified definition introduced by [CBNV13] where each property of a program like instruction mix, working set, ISA, ... is called a *feature*. A set of features is called a *signature*. An approach is called *feature-agnostic*, iff the features composing the signature used cannot be directly mapped onto easily observable properties of the binary run or the system used. In other words, an approach to similarity looking exclusively at execution time would be feature-agnostic, where an approach investigating properties of the binary executed instead would be considered *feature-aware*. We decided to take a feature-aware approach by selecting a signature representative of the underlying architecture and start by implementing a logging system able to gather this data. The result should be able to inform us about the similarity of two given workloads.

We thus define our notion of similarity: Two arbitrary programs μ, ϕ are similar, iff knowledge about the performance of one on a given artifact $\mathcal{A}$ allows us to determine the performance of the other within a factor ϵ. The smaller ϵ, the better our measurement of similarity.

This definition is rather strict and hard to guarantee given the complexity of modern systems as well as the many different bottlenecks they present. It is important to note that output *does not* dictate similarity. Two different implementations of the same problem may generate vastly differing binaries stressing very different subsystems of an artifact.

1.3 Contributions

- We provide a hardware framework capable of logging in-flight and total workload telemetry data based on a defined feature signature.
- We demonstrate the logging capabilities of this framework using industry standard benchmarks.

CHAPTER 2

Past approaches

We are not aware of an implementation specifically generating a modified hardware artifact for this purpose. While tracing hardware can be used for this purpose, the scope of data exposed is often limited as metrics focus on software debugging. While hardware independent metrics can be measured this way, hardware bound data like cache misses would have to be simulated based on the trace data, defeating the idea of an in-hardware approach. Additionally, data generated by tracers necessitate post processing as only the trace in binary is exported and no actual performance data is generated. Enabling tracing is also not transparent to the workload running on the artifact tested. Datalynx aims to be indistinguishable in terms of environment behavior and performance by exporting the performance data via a separate bus (see Chapter 4). Finally, tracers require (depending on the configuration) much larger data rates than exporting consolidated data like this solution.
The basic principals for measuring similarity metrics stay the same to previous attempts without direct hardware support and this thesis pulls heavily from past attempts, especially from the category of feature-aware approaches. This chapter is a short summary of different past attempts at measuring program similarity.

2.1 Feature-agnostic approaches

The simplest feature-agnostic approach is to measure program execution time. This method was chosen by Dujmovic and Dujmovic [DD98] in order to gain insight into how different machines perform. Using this information, a set of machines is compared to each other. Two machines are recognized as equal in computational power if they outperform each other by the same ratio in two different, equally weighted benchmarks. This method of measurement ignores all architectural differences of machines and just focuses on results visible from an outside perspective.
A very similar methodology was also employed in the much newer paper by Cammerota

et al. [CBNV13], where in addition to measured values, machine learning was used to fill out unknowns.

Vandierendonck et al. [VB04] measures execution cycles, rather than time. This tries to compensate for generational clock speed improvements and focuses on architectural changes only. However, like with [DD98], a black box approach is taken. A set of machines is measured and arranged as the n dimensions of a performance vector of a benchmark set. These cycle counts are then transformed into principle components by applying principal component analysis (PCA), which the paper labels *bottlenecks*. Different principal components contribute to the overall performance vector by different amounts, with only a small part of them contributing significantly. These are selected and titled *usage modes*. Four different usage modes are found in the Standard Performance Evaluation Corporation (SPEC) 2000 benchmark suite (SPECint, SPECfp) they examined, which leads them to subset the suite into a set of 9 programs which exercise these usage modes representatively for the entire suite.

These methods have one big advantage as well as one big disadvantage. Evaluating programs like this is simple as virtually no instrumentation is applied to the binary itself. The runtime of the program in question is not modified and no simulation effort is required. Even if the data resolution is low, it is still enough to make certain performance predictions. However, architectural features are completely ignored. Suppose we have a set of machines to determine benchmark similarity but none of these machines feature hardware support for floating point operations. Of the set of examined programs, one happens to be floating point operation heavy and also just so happens to perform similar to one not using floating point operations. When adding a new machine introducing a hardware floating point unit (FPU), performance models cannot predict performance anymore as FPU operations in the binary were not considered as a characterization criterion. Worse still [VB04] might have marked the benchmark for removal, meaning there is a possibility we would have never found the improvement.

We thus conclude that a pure outside-in perspective using a black box model is not sufficient to determine the similarity of two programs.

2.2 Feature-aware approaches

In contrast to the papers presented in Section 2.1, the approaches presented in this section actually consider features exhibited by the programs run. As a hardware artifact consists of several parallel subsystems, workloads are probed in regards to how heavily those subsystems are stressed. This approach naturally requires more probing effort but results in much more fine grained data. Other than the papers in Section 2.1, as long as a certain feature is contained in the signature of a program, two programs cannot be mislabeled similar for that specific feature.

The most prominent publications in this regard are by Phansalkar and Joshi et al. [PJEJ05, JPEJ06] as well as Eckhout et al. [EV03]. While their data sets do vary, all

of them exhibit a clear intent to measure the impact of certain workloads on different subsystems of a microprocessor. The following describes the main subsystems captured and the metrics to capture them between the different publications.

2.2.1 Utilization

All papers mentioned use *instruction mix* as their measurement of utilization distribution between the different evaluated subsystems. [PJEJ05, JPEJ06] measure percentage of computational instructions, load/store operations, as well as control flow operations (branches), while [EV03] additionally splits arithmetical operations into integer arithmetic, logical operations, as well as shift- and byte-manipulation operations.

2.2.2 Control flow behavior

Changes in program flow can introduce processing delays like lost cycles if a branching operation cannot be correctly predicted. For this reason, fine grained knowledge about the branching behavior is important to predict program performance as well as measure similarity of workloads. The information is split up into two basic data points: How often do we need to branch and how accurately can we predict whether we need to branch or not. While all three papers measure the mean distance between consecutive branching operations referred to as *basic block size*, they differ in the assumed model of branch prediction. [PJEJ05, JPEJ06] measure percentage of forward pointing branches, percentage of taken branches, and percentage of forward taken branches in regards to taken branches. Meanwhile, [EV03] employs three different implementations of practical branch predictors, namely a bimodal predictor with 8K entries and 2-bit saturating counters, a gshare predictor also using 2-bit saturating counters and 8ki entries xor-ed with the branching history of the last 12 branches, and a hybrid predictor combining the two and choosing dynamically. Notably, this changes the methodology from purely measuring characteristics of the binary to performance numbers of certain actual implementations.

2.2.3 Instruction level parallelism

Instruction level parallelism (ILP) describes how many instructions could theoretically be executed concurrently. Again, the papers take a different approach in measuring this data point. [PJEJ05, JPEJ06] measure the register dependency distance in 6 distinct windows of 2, 4, 8, 16, 32, and greater than 32 as an indirect measurement of ILP. [EV03] on the other hand assumes a perfect machine with infinite resources and measures ILP this way directly. ILP is useful when designing a superscalar microprocessor, as these benefit from higher inherent parallelism.

2.2.4 Cache and data

While unfortunate control flow breaks can introduce minor stalls, the biggest delay is introduced when the core has to directly access main memory. Because of this, it is

especially interesting to investigate data access patterns to gather insight into optimal caching architectures. [PJEJ05, JPEJ06] record instruction and data *temporal locality* as well as instruction and data *spacial locality* in windows sizes of 16, 64, 256, and 4096 bytes. [EV03] simulates five different cache setups with an 8KB and 16KB direct-mapped cache, a 32KB and 64KB two-way set-associative cache as well as an 128KB four-way set-associative cache, each with a block size of 32 bytes, measuring the miss rate of each setup.

Additionally we want to mention the *data stride* metric proposed by Joshi et.al. [JEBJ08], which represents an attempt to preserve and measure patterns shown by the memory access of a program. The difference in addresses between two consecutive memory accesses is measured. This paper is focused on benchmark generation, which we will further elaborate on in section 2.3. This means the data gathered is not averaged but recorded on a per instruction basis. Strides are characterized as 64-bit windows, where an address difference between 0-63 bytes would be considered stride 0, 64-127 bytes would be stride 1, etc. Results of the paper show that most programs from the SPEC2000 int and fp suits exhibit a single dominant stride access pattern and more than 90% show a regular stride behavior.

2.2.5 Data space reduction

All papers exploring feature-aware design exploration mentioned so far perform principal component analysis to reduce the value space to be explored [PJEJ05, JPEJ06, EV03]. This generates a set of new variables called principal components, where the amount of variance in the data decreases with increasing index. An artificial cut-off is set and all variables of higher index are disregarded. While making clustering simpler and reducing data dimensionality, the output of PCA is dependent on the input data set. It is a relative measure, describing the relations between the data points given. Changing the data set will inevitably change the transformation matrix of input data points to principal components and thus the potentially discarded information. Using PCA would require recalculation of the principal components each time a new data point is added and thus inhibit comparability of results. As we are trying to generate a system where absolute data points can be outputted and easily compared, we decided against performing PCA on the data generated.

2.2.6 Linear model

Lastly, we want to present a simplified linear model that lies somewhat in between a feature-agnostic and feature-aware approach proposed by Saavedera and Smith [SS96]. They propose splitting a program into a set of abstract operations (AbOps), where a performance prediction is made by running a machine characterizer on an artifact to gather data about how fast each AbOp can be executed. Then, the binary in question is analyzed for the quantity of certain AbOps and a simple linear model is applied to calculate predicted performance:

$$T_{\mathcal{A},M} = \sum_{i=1}^{n} C_{\mathcal{A},i} P_{M,i} = C_{\mathcal{A}} \cdot P_M \tag{2.1}$$

Where $C_{\mathcal{A}}$ is a vector representing the quantity of each AbOp in the binary and P_M is the performance vector of the specific machine M (with each dimension being the performance of the respective AbOp).

This approach presents a few downsides however. It assumes a linear model for the entire processor as well as little to new influence from cache misses. Additionally, this approach only works with nonoptimized code of the Fortran language. This model would only work in a rather simple processor design without major caching effects, where a linear approximation would be sufficient to characterize the behavior. This model falls short as soon as state based effects like branch prediction or heavy caching comes into play.

2.3 Related work

Recent efforts in the field of determining program similarity have largely shifted from pure similarity detection of two workloads onto more applied topics.

The already mentioned paper by Joshi et al. [JEBJ08] as well as Ertvelde et al. [VEE10] try to generate benchmarks by deriving characteristics exhibited by a given potentially commercial workload into a much shorter yet similarly performing workload. The program is executed, measured over a similar set of metrics as employed in this thesis and converted into a labeled control flow graph. Each node in this graph represents a basic block within the original workload and contains information about the instructions executed in the corresponding program section. Edges represent branches and are annotated with their transition probability, which is determined by the branch transition rate in the program. The resulting graph is then minimized by reducing the number of loop iterations as well as the occurrence of basic blocks and finally randomly mapped to C statements mapping to similar but not identical instructions when run through a compiler. They were able to show that this workload performs similar to the input workload, while not revealing information about the input itself. This would enable software manufacturers to publish benchmarks derived from their proprietary workloads for hardware optimization without the fear of reverse engineering.

Breughe et al. [BE13] evaluates the effect of different inputs on selected benchmarks from the MiBench and SPEC CPU2006 suites and tries to find a minimum number of inputs to create a representative set in terms of energy-delay product (EDP) of a resulting design. They find that randomly selected inputs have a worst case EDP of 56% on average compared to a design point with the minimal EDP. They showed through their microarchitectural-independent characterization method (Basic Block Vector or BBV selection as introduced in [SPC01]) a worst-case EDP deficiency of 6.7% on average with one and 3.7% with two inputs could be achieved. Min-median-max selection was able to select the design point with minimum EDP by using no more than three inputs.

Lastly, several papers have been published about the issue of plagiarism detection in software. In the age of open source it is rather easy to clone a random repository off of pages like GitHub, obfuscate the code and sell it as commercial software. To counter this practice, several methods have been developed to see through possible obfuscation. Dongjin et al. [KHC+13] focuses on detecting plagiarism in Microsoft Windows applications using their *birthmark*. This birthmark consists of the frequency of system API calls in the binary as well as the frequency of API invocations while running the workload. Nair et al. [NRM20] represent programs as their control flow graph and use a network to estimate the graph edit distance between two program pairs. In this context, graph edit distance refers to the number of operations needed in order to transform one labeled control flow graph into the other. This value is then used as a measurement of similarity between the two given workloads. Marastoni et al. [MGDP18] apply image recognition techniques; program binaries are converted to fixed size images and fed into a network. They recognized that while possible, shallow neural networks optimized for image recognition might not provide optimal performance for the task of determining program similarity. A deep convolutional neural network structure however was indeed able to see through several layers of code obfuscation. The shallow neural net was able to achieve an accuracy of 0.92 at image recognition after training on the MNIST data set, but was only able to achieve an accuracy of 0.03 at determining plagiarism on a simplified data set. The deep convolutional neural net was able to achieve an accuracy of 0.94 on their simplified set and 0.88 on their full data set.

2.4 Summary

A perfect model would be able to predict runtime of an arbitrary workload ϕ on a random artifact $\mathcal{A}$ only by knowing its similarity to a different arbitrary workload μ and its runtime metrics on $\mathcal{A}$. Papers have shown that achieving this feat is possible using different approaches. Both black and whitebox models have shown promise in this regard, however blackbox assumptions do present several downsides like obfuscation of potential changes in hardware design as outlined above.
We have also observed that most whitebox models do measure similar if not identical metrics to evaluate program similarity, with some of them even generating synthetic benchmarks from template workloads using the gathered data. For this reason we have chosen a whitebox approach for this thesis further outlined in Chapter 4.

CHAPTER 3

Platform

3.1 Instruction Set Architecture

We chose the open RISC-V ISA as our target. RISC-V is a free to use instruction set architecture (ISA) which originates from work conducted at Berkley University [His22]. This royalty-free aspect has lead to frequent use in academic papers and emergence of numerous open and closed source hardware designs using the ISA. The architecture is a basic RISC-like load/store architecture featuring an unprivileged and privileged specification [RIS22]. The Pulp Platform, which this thesis is based on uses unprivileged RISC-V and is published under the SolderPad Hardware License v0.51 [Git21].

3.2 Base IP

We base our design on the Pulpissimo microcontroller IP provided by the ETH Zürich and University of Bologna [SRP+18]. The core used in the design is the CV32E40P, formerly known and here (for simplicity) referred to as RI5CY [GST+17]. RI5CY is a 32-bit core with a single-issue, in-order, 4-stage pipeline implementing the RV32IM[F]C instruction set. Floating point support can be enabled and disabled ([F]) depending on requirements towards the core. We will have floating point support enabled as dedicated hardware FPUs have become rather common even in embedded System On Chips (SoCs). RI5CY does support several nonstandard extensions to the RISC-V ISA such has hardware loop, dot product multiplication (also referred to as multiply-accumulate), nonstandard floating point formats, as well as packed SIMD operations.

3.2.1 Pipeline

As seen in Figure 3.1, RI5CY consists of 4 pipeline stages. Stage one is an instruction fetch stage using a 4 entry prefetch buffer as well as a hardware loop (HWL) controller,

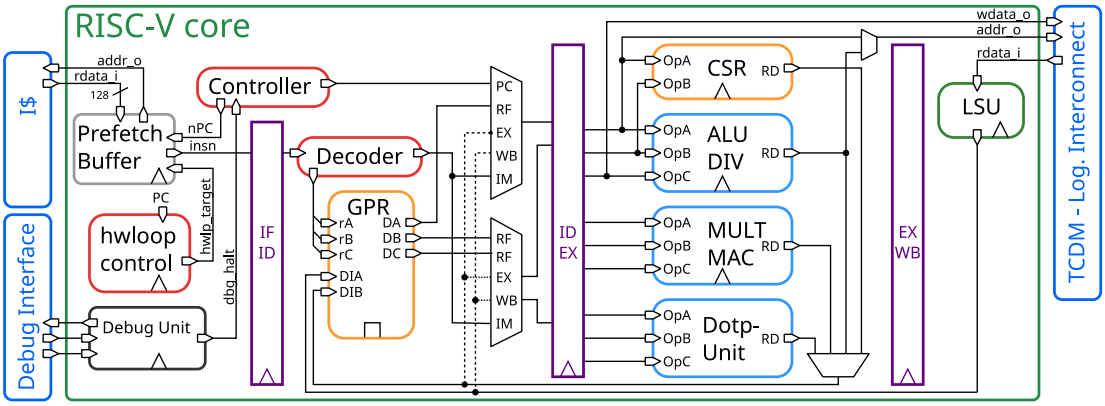


Figure 3.1: Block diagram of the RI5CY pipeline

which we will go into detail about in Section 3.2.3. Stage 2 is the instruction decode stage, containing the RISC-V decoder as well as the register file and HWL registers. Stage 3 is the execute stage featuring an arithmetic logic unit (ALU) for arithmetic and logic operations except multiplication, as well as a multi-cycle multiplication unit which can also perform multiply accumulate operations. Additionally, if enabled, the exec stage also contains the FPU which we will go further into in Section 3.2.4. The 4th and final pipeline stage contains the load/store unit as well as the register write-back. The main design goal of the RI5CY core was to provide a low power, high performance digital signal processing (DSP)-focused core [GST+17], which is reflected in the chosen set of optimizations.

3.2.2 Instruction memory architecture

RI5CY does not feature a full level 1 cache. Instead it contains a 4-instruction deep prefetch buffer inside the instruction fetch stage. This buffer is able to preload up to 4 instructions for execution and is used as a minimalist loop buffer for the HWL implementation described in Section 3.2.3.

3.2.3 Control flow

RI5CY implements a simple *branch-not-taken* prediction strategy. However, in order to reduce branching overhead, RI5CY implements a hardware loop (also called zero-overhead-loops) extension to the RISC-V ISA.
The `lp.setup` instructions allow for set-up of a hardware loop within a single step, additional instructions `lp.start`, `lp.end`, `lp.count`, and `lp.counti` allow for explicit write access to the individual registers used for hardware looping. Subsequent loop iterations are handled by the instruction fetch stage and thus cause no stalls in the pipeline as long as loop instructions fit inside the prefetch buffer. A HWL-based jump is executed when the difference between the program counter and the loop end point stored in the HWL registers is zero, as long as the loop counter has not reached zero. When the

Table 3.1: Configuration of the FPU for RI5CY [MSZB21]

Format	**Implementation** (number of cycles, number of lanes)			
	ADDMUL	DIVSQRT	COMP	CONV
FP32	merged (1,1)	disabled (-,-)	parallel (1,1)	merged (1,2)
FP16	merged (1,2)	disabled (-,-)	parallel (1,2)	merged (1,2)
FP16alt	merged (1,2)	disabled (-,-)	parallel (1,2)	merged (1,2)

counter reaches zero, normal execution is resumed. The HWL controller in RI5CY is configured to support 2 levels of nested loops, as diminishing speedup to area increase was observed at higher levels [GST+17].

Apart from DSP calculations, the inclusion of HWLs has also shown to provide significant speedups for other loop heavy calculations. Experiments by Vreča et al. [VSG+20] have shown that the inclusion of HWLs alone have provided a 29% reduction in clock cycle count compared to a baseline with no optimizations (loop unrolling, HWLs, dot product).

3.2.4 Floating point support

The FPU is enabled for the experiments in this thesis. RI5CY uses FPnew, a highly configurable open-source multiformat floating point unit [MSZB21]. FPnew is split into four different data paths, each of them having configurable datapath depth and width, and supports transprecision computing [MSM+18]. In addition to 32 bit single precision (FP32), and 64 bit double precision (FP64) it supports half-precision (FP16), 16 bit bfloats (FP16alt), as well as an 8 bit custom format (FP8). For RI5CY, only support for FP32, FP16, and FP16alt is enabled. The four data paths implement the functions addition/multiplication (ADDMUL), division/square root (DIVSQRT), comparison/bit manipulation (COMP), and conversion among FP formats/to/from integers (CONV) respectively. These datapaths can be either generated in parallel to support one data format per lane or merged to be multiformat. They can also be disabled completely, if required. For RI5CY, ADDMUL and COMP are enabled in merged mode to save power and area allowing one FP32 operation or a packed vector of either 2 FP16 or FP16alt, COMP is implemented in parallel mode allowing for one FP32 operation or a packed vector of either 2 FP16 or FP16alt, and DIVSQRT is disabled outright (as seen in Table 3.1). All pipelines have a depth of 1 as clock requirements are relatively relaxed for smaller MCUs like RI5CY, so floating point operations can complete within a single cycle. The FPU is integrated into the auxiliary processing unit (APU) port of the execution stage. These details are important for Chapter 4, as this configuration means all FPU operations on RI5CY complete within a single cycle and thus only the percentage of FPU instructions has to be measured.

The FPU is integrated into RI5CY using the cores auxiliary processing unit interface.

3.3 Xilinx ZYNQ-7000 SoC

The hardware used is a ZedBoard by Digilent. It is equipped with a Zynq-7000 All Programmable SoC by Xilinx and natively supported by Pulpissimo. The Zynq-7000 processor features a programmable logic (PL) as well as a Dual ARM Cortex-A9 MPCore hard IP able to run Linux. The hard IP and PL are connected via an AXI bus which can be configured using block diagrams inside the Vivado design environment [Zyn]. An architectural block diagram can be seen in Figure 4.1.

CHAPTER 4

Architecture

In contrast to most approaches presented in Section 2, neither do we use outside-in metrics like execution time, nor do we instrument the binary. The goal was to provide similarity data beyond simple execution time for a given binary which would stay consistent across platforms, as long as the binary itself was unchanged. Additionally, the workload under test should not require any modification which could impact performance and thus skew real world application performance dependent on outside interrupts. We thus concluded the most effective method to be modifying open source hardware and extract data directly from the running core. While this specific implementation is, in the context of all RISC-V MCUs, merely a proof of concept, it can be used to determine similarity of programs and thus representativeness of benchmarks on the specific platform it was implemented on.

We designed an architecture aiming to measure the stress on the same subsystems as [PJEJ05, JPEJ06, EV03] while simplifying certain data points to the specific environment provided by RI5CY. While some data points have been adopted without change, some have been modified as the alternative might be simpler to measure and sufficient for determining similarity on RI5CY. We will justify our choices as well as elaborate on the implementation of all data points read.

The data points are handed from a hardware add-on inside the RI5CY softcore to the Linux instance running on the Dual ARM Cortex-A9 MPCore hard intellectual property (IP).

We thus preset *Datalynx*, a non-invasive data gathering engine implemented as an extension to RI5CY. Datalynx consists of three main parts: An encoder addon to the control and status register (CSR) module inside the RI5CY core called *Enlynx*, an AXI GPIO based connection between the Pulpissimo SoC and the ARM hard IP called *Datalynx bridge*, and finally a Linux based software logger, reading data from the memory mapped GPIO units called *Delynx*.

4.1 Observed features

We attempt to measure the impact of workloads run on the core on different subsystems. In the following we will describe the data points measured by the hardware add-on we call *Enlynx*.

4.1.1 Utilization

In order to gauge utilization of the different subsystems, we have opted to measure

- total number of instructions
- number of multiplication instructions
- number of memory load instructions
- number of memory store instructions
- number of jump instructions
- number of branching instructions
- number of branches taken
- number of hardware loop initializations (further described in Section 4.1.2)
- number of FPU operations.

From this, we are able to calculate the percentage share of each of those instruction types, as well as the share of arithmetic instructions except multiplication, as instrumentation in hardware proved rather difficult.

An instruction is registered every time the decode stage marks an instruction as valid.

A load instruction is registered when the memory protection unit (MPU) sends out a data request, the data request is granted, data write enable is not raised, and the decode stage marks the instruction as valid. A store instruction is registered when the MPU sends out a data request, the request is granted, data write enable is raised, and the decode stage marks the instruction as valid.

We have decided to split multiplication instructions from other arithmetic instructions as the former is handled by a dedicated multiplication unit which can also handle dot product multiplication [GST+17]. While the multiplication unit can handle these operations within a single cycle, we felt the necessity to split these operations for the sake of completeness. A multiplication instruction is registered every time the multiplication unit is invoked and the decode stage marks the instruction as valid. We have opted to calculate the arithmetic instructions as measuring them proved to be more difficult

than expected. Arithmetic instruction count is determined by simply excluding all other instruction types.

A branching instruction is registered when a branching instruction reaches the execution stage and the decode stage marks the instruction as valid. A branch taken is registered when a branching instruction reaches the execution stage, the decode stage marks the instruction as valid, and the execution stage marks a positive branching decision. A jump instruction is recorded when the decode stage encounters a jump instruction and the decode stage marks the instruction as valid. A hardware loop initialization is recorded when the write enable for the iteration counter is raised and the decode stage marks the instruction as valid.

An FPU instruction is recorded when the APU's enable signal is raised by the execution stage and the decode stage marks the instruction as valid.

4.1.2 Control Flow Behavior

To measure the control flow behavior of RI5CY, we record the number of branching instructions, the number of taken branches, the number of HWL initializations, as well as the number of HWL-induced branches.

RI5CY does not use any sophisticated branch predictors but utilizes a simple *branch not taken* approach. For this reason, we can ignore features like branch transition rate [HSF00] for this specific architecture. Occurrence of branching instructions and taken branches are recorded as described in section 4.1.1. For a more generalized approach, a more pattern based method of recording branch behavior might be desirable, as more complex architectures often implement prediction schemes based on patterns in the branching history.

We record a HWL initialization when the HWL counter register is issued a write enabled, meaning the number of iterations is configured. We register a HWL jump each time the HWL controller inside the fetch stage issues a HWL jump.

4.1.3 Instruction Level Parallelism

We have opted to omit ILP as a point of measurement in this implementation, as Pulpissimo is a single-core MCU. Since its bigger brother Pulp [PRL+19] is a multi-core MCU, adding support for ILP measurement might be desired for a future iteration.

4.1.4 Cache and Data

RI5CY does not feature a full level 1 cache. Instead, a 4-instruction-deep prefetch buffer is used to minimize delays towards the decode stage. We attempt to measure the effectiveness of this prefetch buffer by comparing the amount of instructions executed to the amount of instructions requested by the buffer. Because of this implementation, we have opted to omit spacial and temporal data locality, as virtually no locality effects can

be exploited by this system. However, the prefetch buffer still acts as a loop instruction buffer for the HWL system. Thus, a program with a smaller mean instruction count inside `for`-loops should see a better instruction-to-load ratio overall.

An instruction request is recorded when the prefetch buffer sends an instruction request to the MPU and the MPU grants the request.

4.2 Communication with the Linux system

Communication with the Linux system is achieved via a AXI GPIO unit IP utilizing the Programmable Logic to Memory Interconnect seen at the bottom right of Figure 4.1. These units have two input ports on the PL side with a width of 32 bit each. The GPIO units communicate with the Zynq-7000 processing system via an AXI crossbar module. The GPIO units, the AXI crossbar unit, as well as the Zynq-7000 processing system are provided by Xilinx as IP cores. As mentioned before, the inputs of the GPIO are mapped into memory by the Linux operating system and the memory map is visible in Table 4.1. The block diagram is visible in Figure 4.2 and we call the displayed component linking the RI5CY and Linux part of the Datalynx system the Datalynx bridge.

4.2.1 Additional flags

Additionally to the described main features, the data link contains several other flags to indicate the state of the system. An overflow vector indicates the saturation of each counter where one bit signals the saturation of its respective counter. Lastly, the end of program is indicated to the logger on the Linux side by the `eop` flag.

4.3 Enlynx

Events are registered via the Enlynx add-on integrated into the CSR unit. It is implemented similarly but separate to the performance counter (PC) module. Unlike the PC module, the counters are not implemented as readable registers but as configurable width counters only exposed via the Datalynx bridge. As the limit of the Xilinx provided AXI GPIO 2.0 units is 32 bits, we have opted for 32 bit unsigned integer counters to make the setup as simple as possible as well as maximize the value range. The GPIOs are memory mapped in Linux starting at address `0x4120_0000`. Table 4.1 shows the memory layout as observed in Linux. Counter width can be configured via parameters in the `xilinx_pulpissimo.v` top module by changing the `NLYNX_COUNTER_WIDTH` parameter. Increasing the counter width beyond 32 bit would require reconfiguration of the Datalynx block diagram, as two GPIO ports would be needed for a single event counter.

Like the PC module, the counter can be enabled and disabled via a write to a control register at address `0x7A2`. This is to crop out any instructions executed by the runtime prior to payload execution. The register holds two bits of information. Bit `0` enables

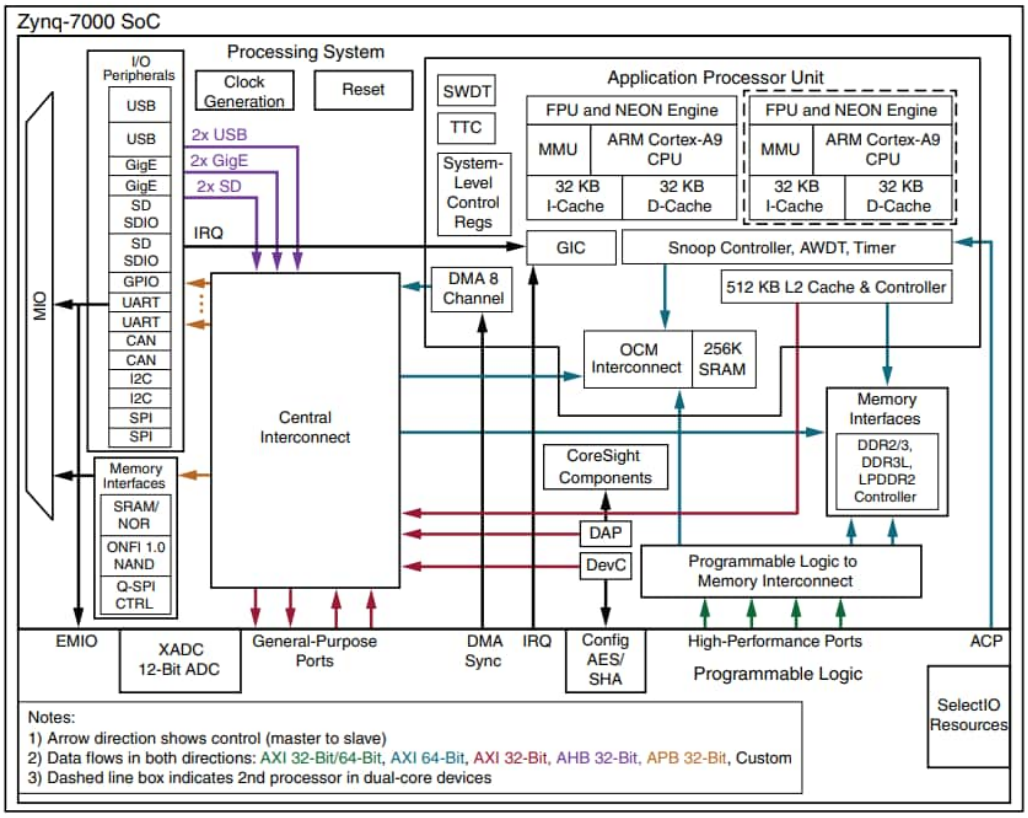
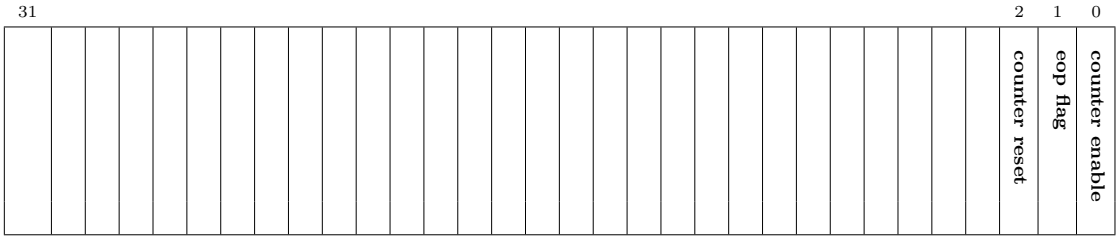


Figure 4.1: Block diagram of the Xilinx Zynq-7000 All Programmable SoC [Zyn]

Table 4.1: Memory map for the event counters in Linux

Counter	**GPIO unit #**	**Memory address**	**Description**
`instr_cnt`	0.1	`0x4120_0000`	Instruction count
`load_cnt`	0.2	`0x4120_0008`	Number of load instructions encountered
`store_cnt`	1.1	`0x4121_0000`	Number of store instructions encountered
`alu_cnt`	1.2	`0x4121_0008`	Number of arithmetic instructions excluding multiplication encountered
`mult_cnt`	2.1	`0x4122_0000`	Number of multiplication instructions encountered
`branch_cnt`	2.2	`0x4122_0008`	Number of branching instructions seen
`branch_taken_cnt`	3.1	`0x4123_0000`	Number of taken branches
`fpu_cnt`	3.2	`0x4123_0008`	Number of FPU instructions
`jump_cnt`	4.1	`0x4124_0000`	Number of jumps see
`hwl_init_cnt`	4.2	`0x4124_0008`	Number of HWL initializations
`hwl_jump_cnt`	5.1	`0x4125_0000`	Number of HWL induced jumps
`inst_fetch_cnt`	5.2	`0x4125_0008`	Number of instruction fetches by the prefetch buffer
`cycl_wasted_cnt`	6.1	`0x4126_0000`	Number of cycles wasted
`eop`	7.1	`0x4127_0000`	End of program flag
`overflow`	7.2	`0x4127_0002`	Overflow vector

Table 4.2: CSR layout for the enlynx control register



Bit #	R/W	Description
0	R/W	**Counter enable:** Activate/deactivate all event counting. If this bit is 0, all event counting is disabled. After reset, this bit is not set.
1	R/W	**eop flag:** Set the counter to total mode and indicate end of program. If this bit is not set, counter values outputted to the Datalynx bridge indicate the metrics of the previous instruction section. Metrics are updated every time a new section is completed. If this bit is set, total metrics since the last reset are outputted to the Datalynx bridge. After reset, this bit is not set.
2	W	**Counter reset:** Reset all internal counters of the Enlynx add-on. This bit will always return 0 when read. Setting the bit resets the counters once.

or disables the counter. When the flag is raised, the global counter inside Enlynx will increase at every positive clock edge for events which have a raised flag. This information is not be visible at the GPIO output right away. Bit `1` is connected to the `eop` flag at the output. The reset value for the control register is `0x0000_0002`. Additionally, Bit `2` will always display `0` on read, but setting to `1` will reset the internal counters.

The Enlynx module has two modes: section mode and total mode. Section mode is active as long as the `eop` bit inside the CSR has not been set. In section mode, the GPIOs show the counter values of the last 65536 instructions executed in the core. This window size can be adjusted by changing the `NLYNX_SECTION_SIZE` parameter in the `xilinx_pulpissimo.v` top module. Window sizes can be set to powers of 2, where the parameter indicates the exponent. After a reset, data will only be visible after the counter has been enabled and the amount of instructions given by the window have been completed. Furthermore, there is no indication for data change on the output. The values are collected by the Delynx decoder on time based approach and are solely meant to visualize behavioral changes of programs over time. As the metrics recorded have been chosen to be able to represent overall behavior of a program, these data points were mostly meant for visualization purposes. A further study might explore possible effects

these changes could have like power draw or heat dissipation, but we expect them to be minor overall in the context of embedded devices.
Total mode is active as soon as the `eop` flag has been set. The GPIOs show global counters since the last reset of the core. If a counter has been saturated since the last reset, the output will show `0xFFFF_FFFF` and the corresponding bit of the overflow vector is set.

The overflow vector is tied to the global counter in the module. Although a sectional counter might not be saturated, data is to be treated invalid as soon as the overflow flag of a specific counter is set.

Payloads are not intended to read from or write to the Enlynx CSR. Accesses to the register are made by the Pulp runtime only just before and after payload execution.

4.4 Delynx

The Delynx decoder is a simple program running on the programmable software (PS) side of the SoC. It is executed by the makefile initializing the tests and continually reads the data provided by the Datalynx bridge. The read intervals are controlled on a time basis rather than an instruction basis to represent the change of program behavior over time. This represents a possible point of data loss however. As we will see in Chapter 7, data resolution appears to be high enough for potential data loss to be negligible. The `eop` flat is checked before and after data has been read, to ensure global data is not accidentally read as a local value. After a transition on the `eop` flag has been seen, global data is read, saved, and the decoder terminated. Data is saved into a corresponding `.csv` file containing time stamps and values read.

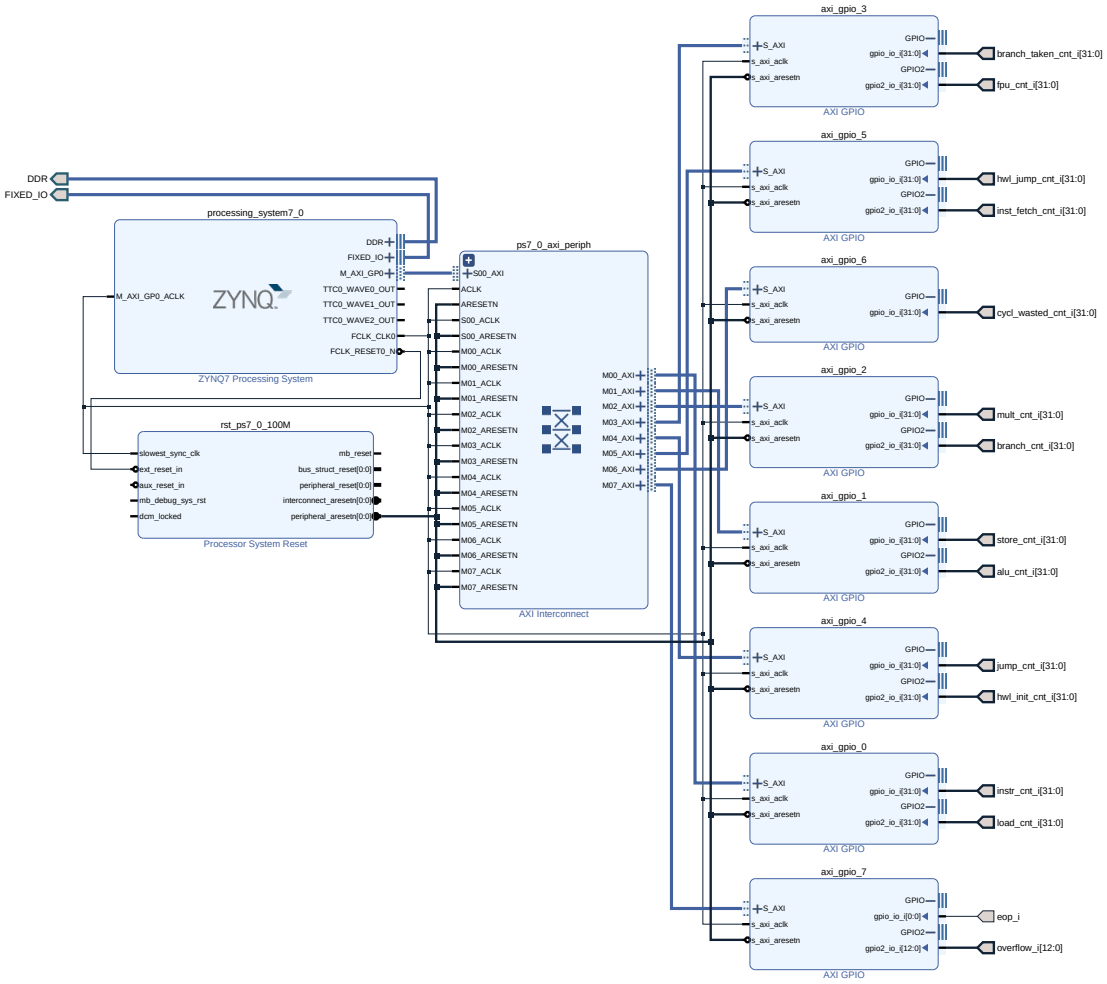


Figure 4.2: Block diagram of the datalynx bridge

CHAPTER 5

Impact on Hardware

We have observed an expected increase in resource utilization on the SoC. Datalynx requires additional look-up tables (LUTs), LUTRAMs, flip flops (FFs), as well as one additional global clock buffer (BUFG) as seem in Table 5.2. According to the timing analysis performed by Vivado, our worst slack stayed about the same slightly improving from 1.316ns to 1.767ns with the total negative slack being 0ns. Hold slack also slightly improved from 0.054ns to 0.055ns with total negative hold slack again being 0ns. As we have not changed the implementation of the core itself, the additions should be completely transparent to a workload run on the processor, as long as the core frequencies are unchanged. While the critical path for setup slack has not changed in the modified design, the path for hold slack actually has, as can be seen in table 5.1. However, this occurs within the block diagram wrapper generated by Vivado and does not worsen overall hold slack performance significantly. Additionally, this change is likely caused by high utilization of the chip overall and the resulting less than ideal routing results. We thus see this change in critical path as a minor issue albeit something to be aware of.

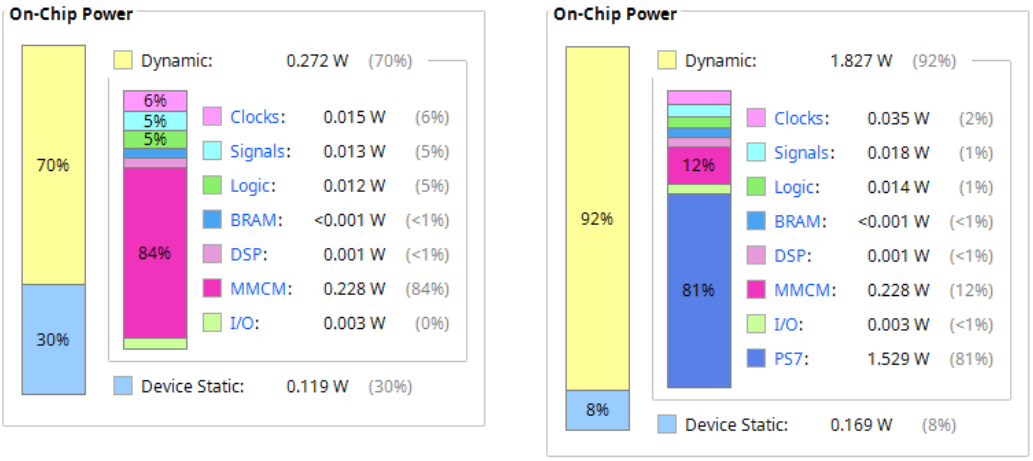
Of note is the increase in power draw according to Vivado. This metric is far from accurate, as these power metrics are dependent on Vivado's randomized optimizations, meaning different runs will yield sightly different reported power characteristics. Additionally, due to the nature of this implementation, higher power draw is expected as two additional cores have been enabled on the chip. A different implementation might choose a more minimalist method for offloading data from the chip, significantly lowering power draw impact.
In our synthesis run, Vivado estimated about 0.391W power draw for the unmodified and 1.996W power draw for the modified design.
As seen in Figure 5.1, the share of dynamic power losses increases significantly with the inclusion of the processing system of the Zynq-7000 (PS7). This block alone makes up about 81% of power drawn in the modified system. When excluded, power consumption only rises by 76mW compared to the original design. This is still a skewed value

Table 5.1: Critical paths for setup and hold slack in unmodified and modified design

Design	Path	
Unmodified	From	i_pulpissimo/soc_domain_i/pulp_soc_i/jtag_tap_top_i/tap_top_i/ td_o_reg/C
	To	pad_jtag_tdo
	Setup	1.32ns
Modified	From	i_pulpissimo/soc_domain_i/pulp_soc_i/jtag_tap_top_i/tap_top_i/ td_o_reg/C
	To	pad_jtag_tdo
	Setup	1.77ns
Unmodified	From	i_pulpissimo/soc_domain_i/pulp_soc_i/soc_peripherals_i/i_apb_adv_timer/u_apb_if/r_timer1_th_reg[9]/C
	To	i_pulpissimo/soc_domain_i/pulp_soc_i/soc_peripherals_i/i_apb_adv_timer/u_tim1/u_counter/r_start_reg[9]/D
	Hold	0.05ns
Modified	From	datalynx_wrapper_i/datalynx_i/axi_gpio_4/U0/ip2bus_data_i_D1_reg[11]/C
	To	datalynx_wrapper_i/datalynx_i/axi_gpio_4/U0/AXI_LITE_IPIF_I/ I_SLAVE_ATTACHMENT/s_axi_rdata_i_reg[20]/D
	Hold	0.05ns

Table 5.2: Resource utilization unmodified vs. modified

Resource	Available	Utilization unmodified	Utilization % unmodified	Utilization modified	Utilization % modified	Change
LUT	53200	43502	81.77%	46432	87.28%	+6.7%
LUTRAM	17400	12	0.07%	74	0.43%	+516.7%
FF	106400	22526	21.17%	28214	26.52%	+25.3%
BRAM	140	128	91.43%	128	91.43%	+0%
DSP	220	12	5.45%	12	5.45%	+0%
IO	200	38	19.00%	38	19.00%	+0%
BUFG	32	9	28.13%	10	31.25%	+11.1%
MMCM	4	2	50.00%	2	50.00%	+0%



(a) Unmodified design (b) Modified design

Figure 5.1: Estimated power consumption according to Vivado

however, as other IPs by Xilinx, like the GPIOs and AXI crossbar unit have not been excluded. Including the PS7 for the modified design leaves us with an estimated junction temperature of roughly 48.0°C for the modified and 29.5°C for the unmodified design, resulting in a 37.0°C and 55.5°C temperature margin respectively.

CHAPTER 6

Benchmarks

Over the past decades, a handful of benchmark suits have managed to establish themselves as the de facto industry standard for embedded benchmarking. These suites differ heavily from their desktop counterparts given the stark difference in environment they operate in. Most of these suites have emerged in the last 20 years and newer ones often reuse workloads from preexisting suites. The main differences between embedded benchmarks and desktop computing benchmarks are storage and system requirements. Desktop computing benchmarks assume an operating system to be present on the machine and thus perform syscalls and expect commodities like a file system. Embedded benchmarks need a small storage footprint to be able to even fit on the heavily restricted flash space of embedded processors. Additionally, while desktop benchmarks are mostly focused on raw performance, embedded benchmarks often consider additional performance metrics such as power to throughput. In the following we will present the list of benchmarks we selected to test our framework with.

6.1 Coremark

Coremark is a purely synthetic benchmark aimed at providing a single performance score for as many embedded systems as possible. From the list presented here, Coremark is the only fully synthetic benchmark, which makes the comparison against an actual workload much more interesting. The benchmark consists of three main segments, which are list processing, matrix processing, and state machine processing. List processing iterates through a list of data containing either precomputed values or algorithm invocation instructions, where the list itself is also searched, inversed and sorted. Matrix processing mainly strains the compute engine of a core, performing matrix operations which cannot be computed at compile time. These operations are also meant to test the efficiency of a core's compiler optimizations as well as its handling of tight loop operations and Single Instruction, Multiple Data (SIMD) instructions. The state machine processing segment

Table 6.1: Applications contained in the different sub-suites of MiBench [GRE+01]

Auto./Industrial	Consumer	Office	Network	Security	Telecom.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	
	typeset				

is implemented mainly by using `if`-statements, while the list processing portion employs `switch`-statements in the source code [GL12].

Coremark's instruction profile also strongly varies over time [GL12], which is reflected in sectional data as explained in Section 4.3 and can be seen in Chapter 7.

6.2 MiBench

MiBench came about in 2001 and was a response to SPEC on the desktop side as a standardized benchmark computing suite meant for embedded applications. Its intention was to assemble a portable suite of open source software able to run on as many systems as possible and to be as representative as possible [GRE+01]. In contrast to Coremark, MiBench is a compilation of actual problem solving applications and does not contain purely synthetic elements. MiBench is separated into the categories Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. Each category consists of a set of programs representative of applications possibly run on processors of said category at the time of benchmark design as seen in table 6.1. [GRE+01] goes on to show that MiBench has a higher variation in instruction profiles compared to its desktop counterpart SPEC2000. Thus, selecting not only the right sub-suite, but also the right program(s) from the chosen sub-suite(s) poses an interesting challenge to solve. However, the benchmark suite also assumes a host operating system for most of its programs [PHB13]. This is a rather steep constraint and applications with that requirement will not be tested on our platform.

6.3 BEEBS

BEEBS' main focus is the exploration of energy consumption. The corresponding whitepaper was published in 2013 and is designed for embedded applications. BEEBS is a conglomerate of workloads cherry picked from already existing benchmark suites with a main focus on providing applications big enough to be representative while setting as little requirements for the host platform as possible (storage requirements, file system,

Table 6.2: Applications contained in BEEBS, their origins, and their categorization [PHB13]

Name	Source	B	M	I	FP	Category
Blowfish	MiBench	L	M	H	L	Security
CRC32	MiBench	M	L	H	L	Telecom.
Cubic root solver	MiBench	L	M	H	L	Automotive
Dijkstra	MiBench	M	L	H	L	Network
FDCT	WCET	H	H	L	H	Consumer
Float Matmult	WCET	M	H	M	M	Automotive, consumer
Integer Matmult	WCET	M	M	H	L	Automotive
Rijndael	MiBench	H	L	M	L	Security
SHA	MiBench	H	M	M	L	Network, security
2D FIR	DSPstone	H	M	L	H	Automotive, consumer

...). They group the applications contained into four different categories corresponding to strain exerted on different subsystems of a core under test. Each of the categories, integer operations, floating point operations, memory access intensity, and branching frequency, are marked with either low, medium, or high [PHB13]. The benchmarks were mostly sourced from the MiBench suite [GRE+01] with 3 applications being derived from the WCET suite [GBEL10] and one from DSPStone [ZIV94], as seen in table 6.2. The table also shows their categorization according to the grouping criteria laid out by [PHB13]. The resulting set was tested on different architectures representing different tiers of processors. As the main metric to be observed is power consumption, a memory access on an implementation featuring a cache uses a different amount of energy than the same access on an implementation without one. The applications were validated on a Cortex-M0 processor, a simple single core implementation, an XMOS L1, an event driven platform with eight hardware threads, and Epiphany, a super scalar 16-core processor. The benchmarks were also profiled these three platforms, to ensure that each workload was different enough to all other workloads in order to minimize overlap and redundancy within the suite. [PHB13].

We only use a very limited subset of BEEBS due to availability of ports on the tested platform.

6.4 EmBench

EmBench is an open source benchmarking suite created in 2019 as the one suite to measure them all. The intention was to displace benchmarks previously tagged as industry standard by assembling a free and open-source suite comprised of real world programs maintained by a central organization [Pat19]. The code is available on their official website [Emb], at version 1.0 as of writing, and consists of 19 benchmarks. These benchmarks were largely sourced from MiBench [GRE+01] (see Section 6.2),

Table 6.3: Benchmarks run per suite

Coremark	**MiBench**	**EmBench**	**BEEBS**	**Custom**
Coremark	basicmath	aha-mont64	crc32	Dhrystone
	bitcount	crc32	cubic	median
	dijkstra	cubic	dijkstra	multiply
	fft	edn		rsort
	qsort	huffbench		sort
	stringsearch	matmult-int		spmv
	susan	minver		towers
		nbody		vvadd
		nettle-aes		factorial
		nettle-sha256		nqueens
		nsichneu		tak
		picojpeg		
		qrduino		
		sglib-combined		
		slre		
		st		
		statemate		
		ud		
		wikisort		

BEEBS [PHB13] (see Section 6.3), and TACleBench [FAH+16]. EmBench reports the geometric mean compared to a reference system as well as standard deviation as the single performance score. Additionally, code size of the target platform in regards to the reference system are also reported. Quite fittingly, RI5CY is the reference system for EmBench [Pat19].

6.5 Summary

We use the benchmarking suites presented above to test our hardware framework. Due to availability and portability, we were not able to test every single program in each suite, however all programs tested can be seen in Table 6.3. Additionally to the suites presented above, other benchmarks were also run due to ports for the platform being readily available. They have been added in the *Custom* category.

CHAPTER 7

Results

We were able to successfully generate performance profiles for the workloads described in Section 6.5. The most noteworthy results are presented and analyzed in the section bellow. Graphs for the workloads we have not presented specifically in this chapter can be found in Appendix B.

7.1 Instruction distribution

The workloads vary highly in instruction distribution, however for most of them arithmetic instructions make up the majority of instructions in many workloads (see Figure 7.1). Exception to these rules are `embench edn`, `embench matmult int`, `embench nsichneu`, and `riscv vvadd` with load instructions as the dominant type; `embench statemate` with store instructions as the dominant type; as well as `riscv median` with branching instructions as the dominant type. As can be seen in Figure 7.1, the difference between the most favored category and the runner up appears to be rather large on average. Indeed, the mean difference between the first two categories over all 39 workloads is 32.11% points, where only 7 workloads have a disparity of less than 10% points, with 25 having a disparity of over 20% points and 17 having a disparity greater than the mean. This indicates that each workload strongly prefers one type of instruction with arithmetic instructions being the clear overall favorite.

Moreover, we can see that almost no workload we tested uses FPU instructions, with `embench minver` being the only significant contender with a share of 13.8%. Only `mibench-automotive susan` and `mibench-telecomm FFT` even register with a share of 0.21% and 0.01% respectively.

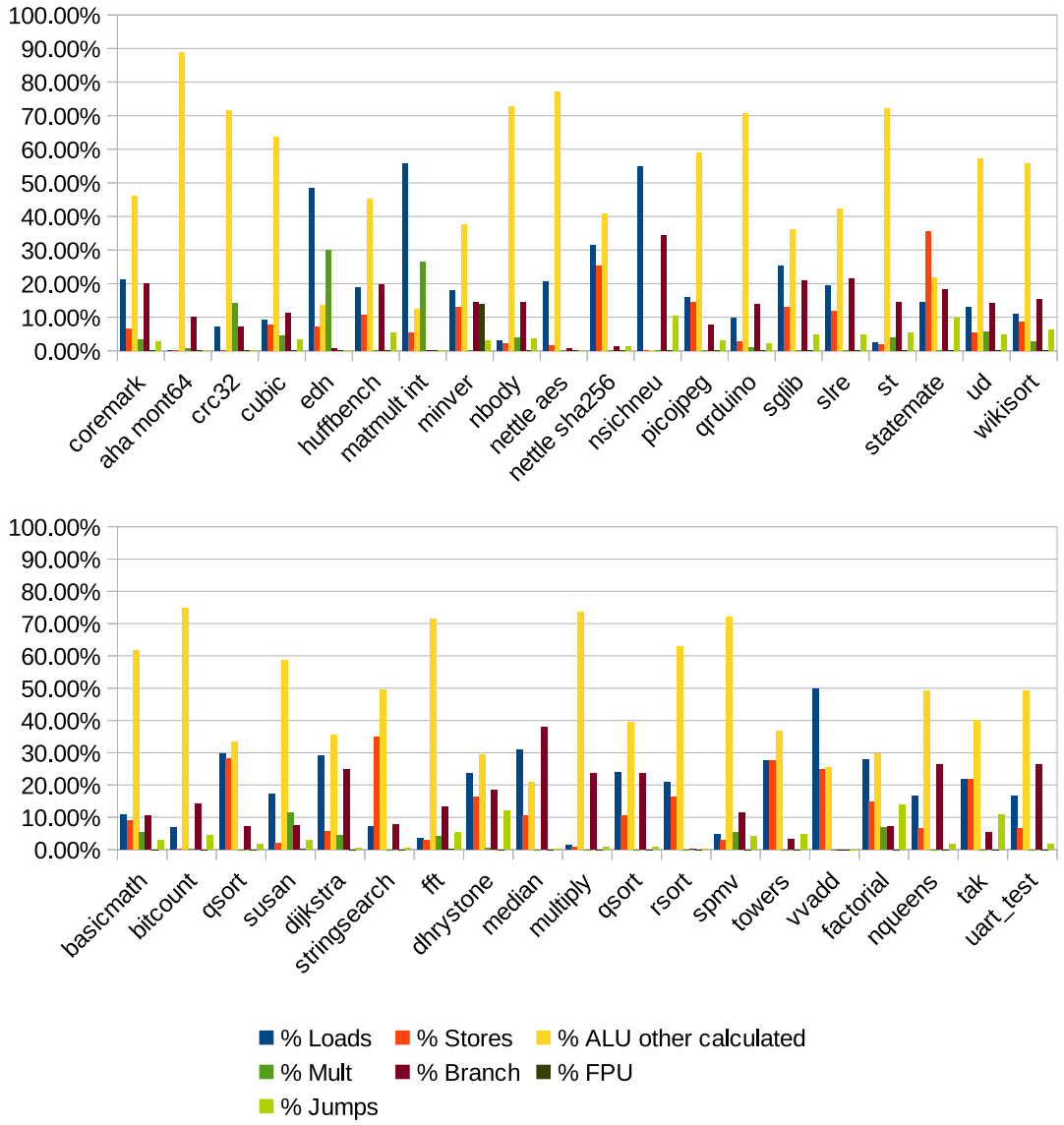


Figure 7.1: Overall instruction distribution of all tested benchmarking workloads

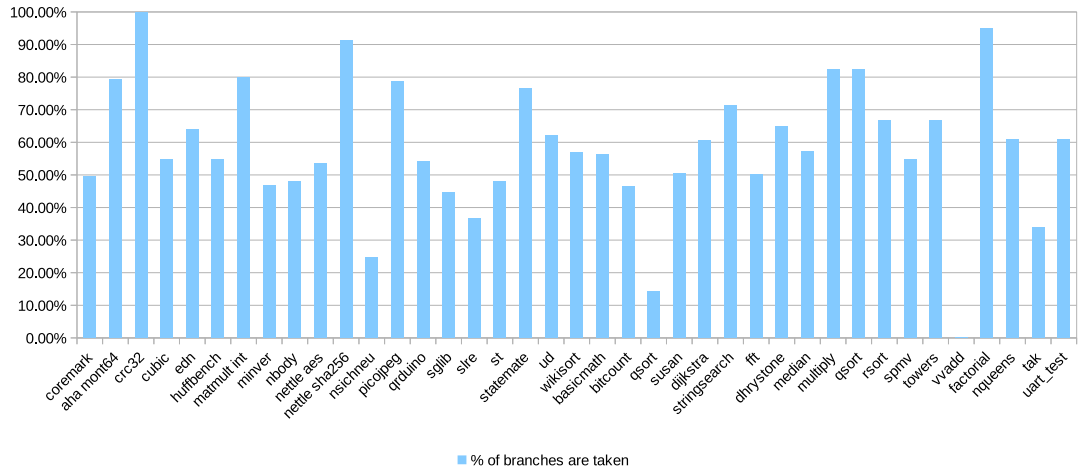


Figure 7.2: Overall branch taken behavior of all tested benchmarking workloads

7.2 Control flow behavior

As seen in Figure 7.2, a vast majority of benchmarks take more than 50% of branches. Given RI5CY's *branch-not-taken*-strategy, this leads to a high percentage of failed branch predictions. Only 10 workloads take less than 50% of branches.

Not all tested workloads employ HWLs. In workloads which do use HWLs, usage modes vary a lot depending on the implemented algorithm. An extreme example of HWL use is `embench crc32` with a mean distance between initializations of $39 \cdot 10^6$ as well as a mean iteration count of $2.7 \cdot 10^6$ as can be seen in Figure 7.3 which is by far the highest number in both distance between initializations and iteration count. We suspect there to be a correlation between these two metrics in extreme cases like this, as no further HWL initialization happens during the run of the loop (thus inflating mean distance between HWL initializations). More work would be needed to determine whether this correlation is an issue for similarity assessment and, if so, how it could be mitigated.

7.3 Instruction load efficiency

As described in Section 7.2, a majority of workloads have a branch taken rate of higher than 50%. The logical assumption would be a higher load coefficient and thus lower instruction load efficiency in these workloads. We have found the relation to be more complicated however. In fact, all five programs with a load coefficient less than 1 (meaning less instructions loaded from level 2 than executed thus higher instruction load efficiency) have a branch taken percentage well above 50%, as can be seen in Figure 7.4 (we intentionally excluded `tak` included in the *custom* set here, as it only executes 2555 instructions and is thus too small to draw conclusions from). A high branch taken rate does not necessitate poor instruction load efficiency. A low branch taken rate does not

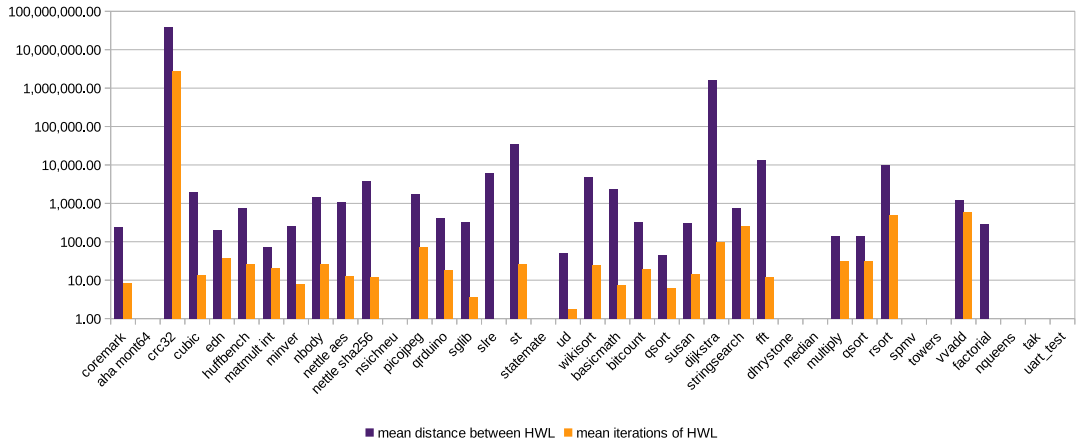


Figure 7.3: Overall hardware loop behavior of all tested benchmarking workloads

necessitate a high instruction load efficiency either. `embench nsichneu` has a branch taken rate of 24.6%, yet a load coefficient of 1.55.

We suspect loop length to play a considerable role in determining the instruction load efficiency in regard to the harsh constraint of a 4 instruction prefetch buffer. `embench nettle aes` and `embench nettle sha256` have a branch taken rate of 53.68% and 91.22% but a fetch coefficient of 0.91 and 0.71 respectively. `embench nbody` is similar to `embench nettle aes` in all metrics, yet has a significantly higher load coefficient. This suggests that `embench nettle aes` has a smaller loop body size than `embench nettle aes` overall and therefor requires less instruction fetches from level 2. Additionally, we suspect instruction alignment to play a large role as well, as adding single instructions at the beginning and end of functions was able to change benchmark performance significantly. Further work would be required to confirm this hypothesis.

7.4 Temporal behavior

Lastly, we have identified three major types of workloads given their temporal behavior. Benchmarks below a certain length have been excluded from this categorization as they did not produce adequate data for analysis. The graphs of those which have been analyzed can be found in Appendix B. We have defined three major groups from the generated data, ranging from completely static to extreme behavioral variation or oscillation.

Group 1 does not change its behavior significantly over the duration of the benchmark. All metrics are unchanged over the course of the entire run. One example for group 1 is Dhrystone as seen in Figure 7.5. The HWL graph for Dhrystone is missing, as Dhrystone does not utilize HWLs. Dhrystone exhibits a branch taken rate in the middle- to high range compared to other workloads, as can be seen in Figure 7.2, with ALU instructions being the prevalent type. What is particularly interesting about Dhrystone is the completely static nature of its metrics. Over the entire benchmark duration of

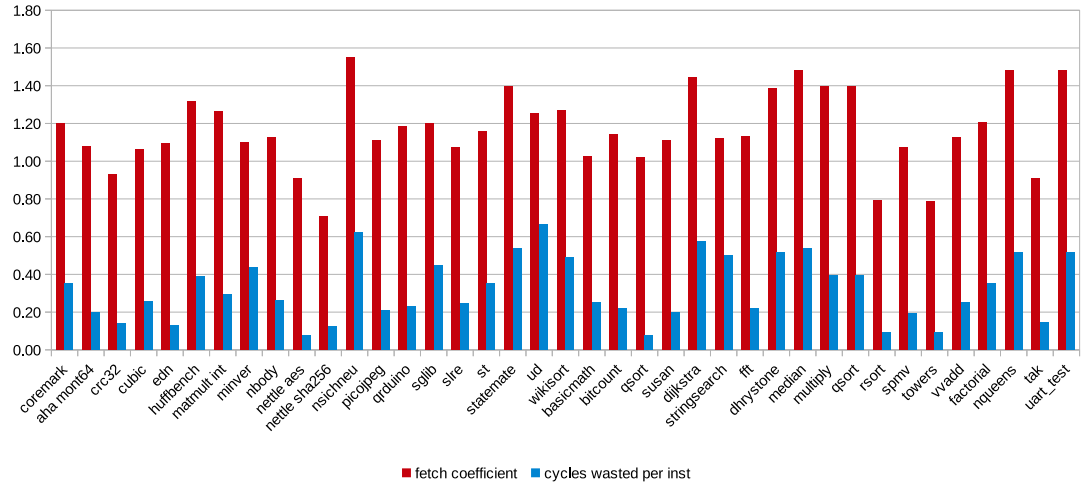
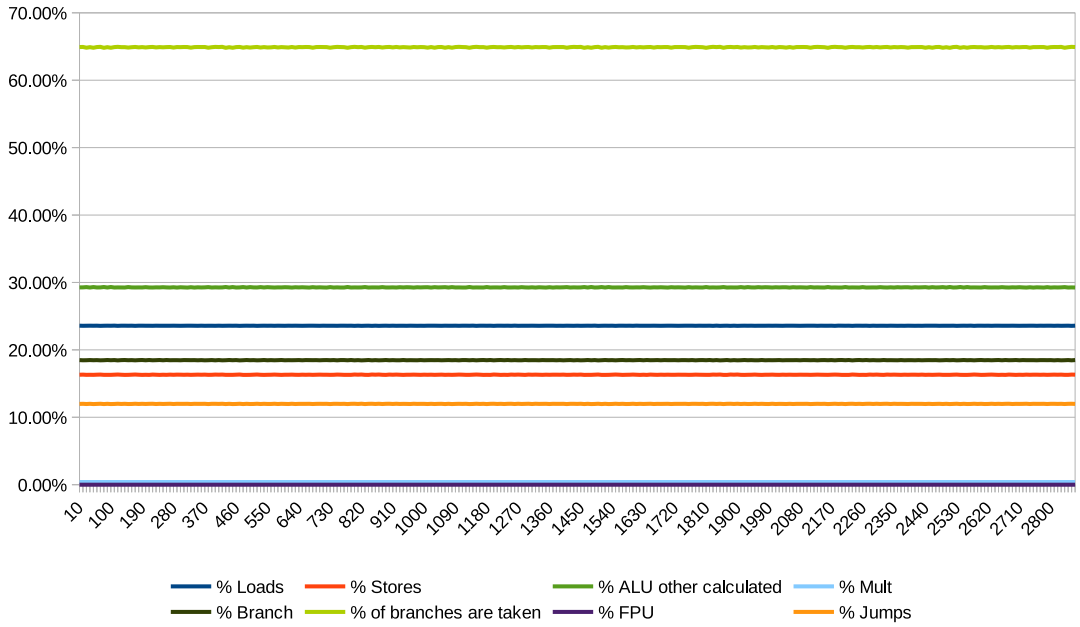


Figure 7.4: Overall load coefficient and cycles wasted per instruction of all tested benchmarking workloads

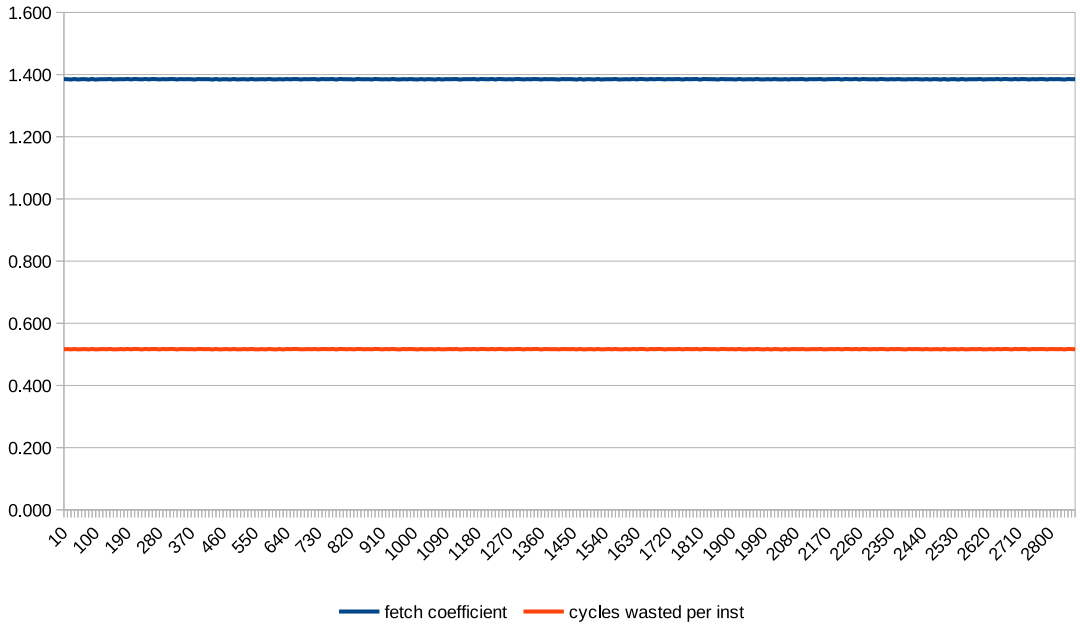
about 2.9 seconds behavior does not change, indicating possibly a short workload that is repeated until an end condition is met. This program loop is very likely a `while` loop with a non precomputable iteration count, as no HWLs are used (indicated by the missing HWL graph).

Group 2 shows sectional behavioral variation, where sections have relatively stable behavior with stark changes between sections. An example for group 2 is `mibench-automotive bitcount`, the temporal behavior graphs of which can be seen in Figure 7.6. The workload displays 7 distinct regions, which hint at different program loops being executed. All regions except the first and last display little to no change in metrics with short initialization phases at the transients. The first 6 sections appear to be implemented using `while` without precomputable iteration counts not unlike Dhrystone, as no HWLs are used. Section 7 does seem to use precomputable loops, as the usage of HWLs is indicated here (see Figure 7.6b).

Group 3 fluctuates strongly during the entire duration of the benchmark. Examples for group 3 are the synthetic Coremark, as can be seen in Figure 7.7, as well as `embench sglib-combined`, as seen in Figure 7.8. Both see more or less drastic variation within all metrics measured, with `embench sglib-combined` displaying behavior reminiscent of oscillation. Coremark seems to exhibit repetition of behavioral patterns, as a zoom into the graph reveals relatively periodic sections, which does match the makeup of Coremark outlined in Section 6.1. `embench sglib-combined` also displays strong variation of metrics, however the variation range is much less drastic than with Coremark. The benchmark displays 4 different sections, which are best exemplified by Figure 7.8b. When the initialization distance between HWLs drops, so does the variation of share of ALU instructions. While `embench sglib-combined` has a relatively stable fetch coefficient around 1.2, Coremark does fluctuate wildly between 1.15 and 1.3.

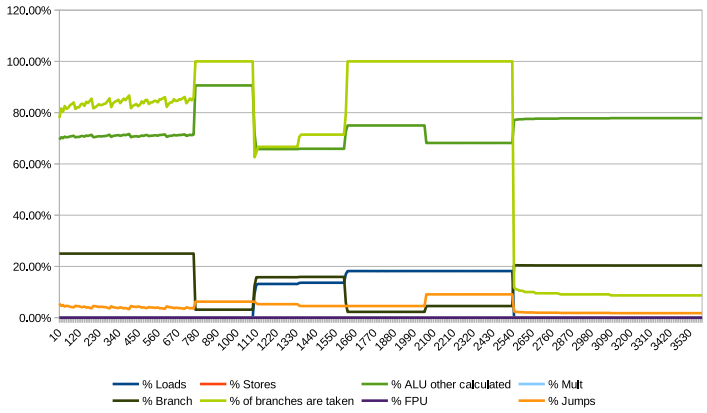


(a) Instruction behavior over time (ms)

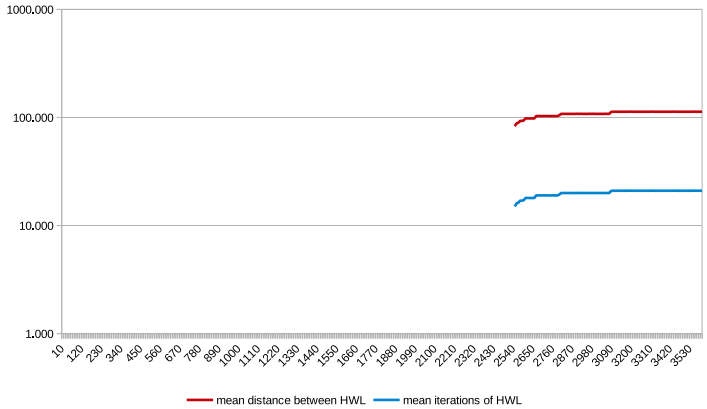


(b) Fetch coefficient and cycles wasted per instruction over time (ms)

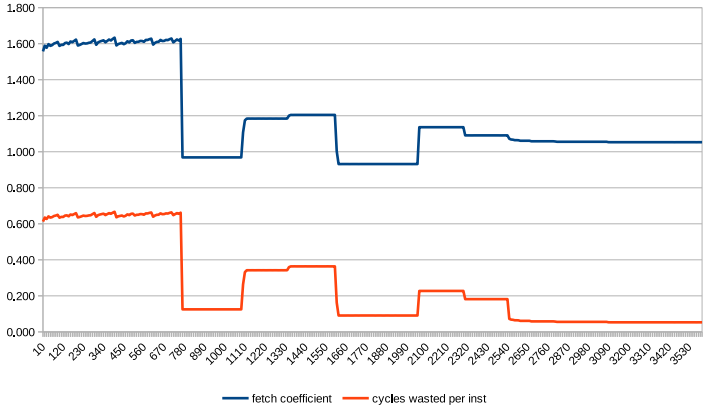
Figure 7.5: Performance over time: Dhrystone



(a) Instruction behavior over time (ms)

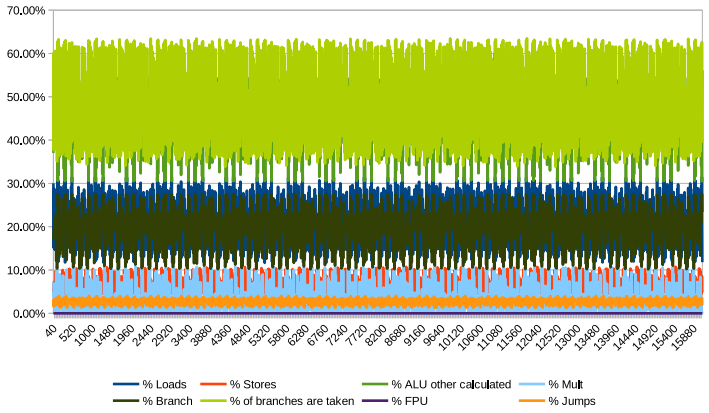


(b) HWL behavior over time (ms)

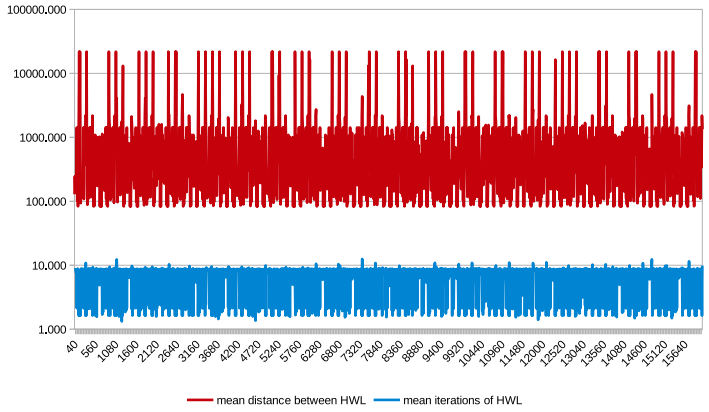


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

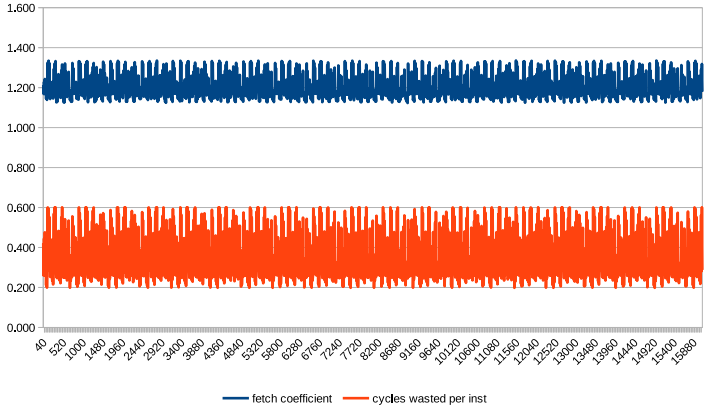
Figure 7.6: Performance over time: `bitcount`



(a) Instruction behavior over time (ms)

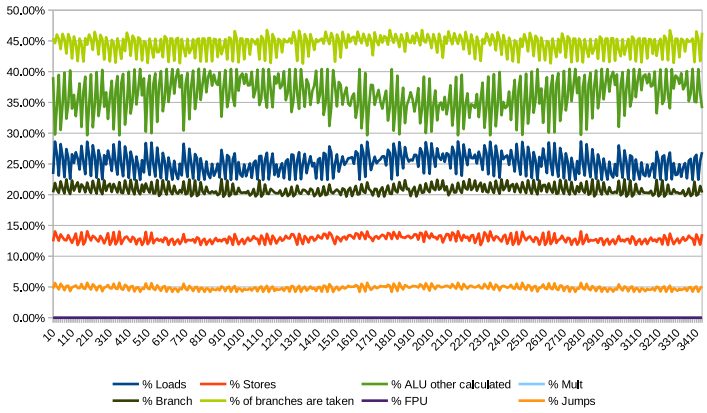


(b) HWL behavior over time (ms)

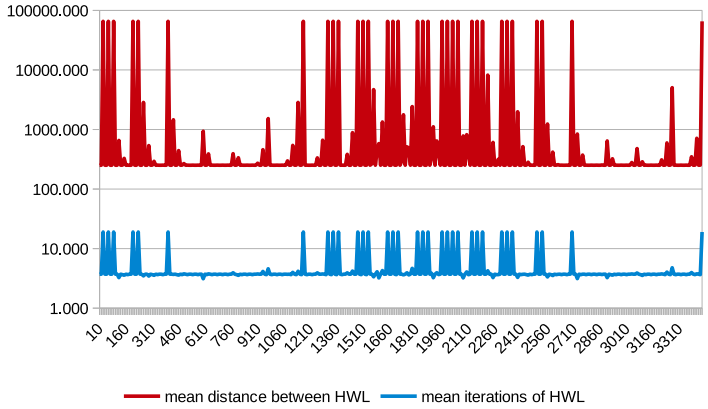


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

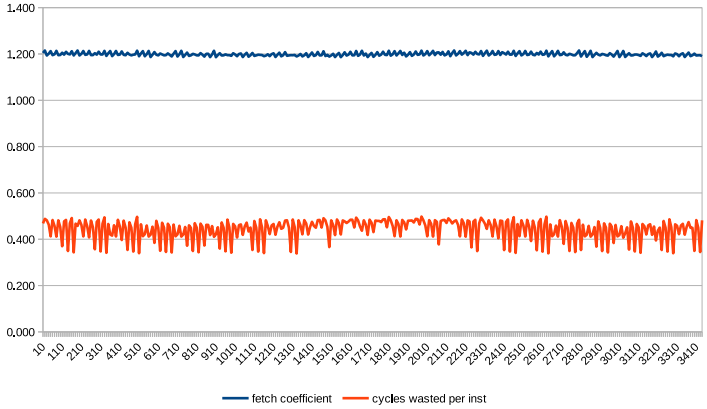
Figure 7.7: Performance over time: Coremark



(a) Instruction behavior over time (ms)



(b) HWL behavior over time (ms)



(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure 7.8: Performance over time: `sglib-combined`

Table 7.1: Group categorization of tested workloads

Group 1	Group 2	Group 3
embench aha-mont-64	mibench basicmath	mibench dijkstra
embench crc32	mibench bitcount	embench cubic
embench edn	mibench susan	embench huffbench
embench minver	mibench fft	embench matmult-int
embench nsichneu		embench nbody
embench statemate		embench nettle-aes
embench ud		embench nettle-sha256
Dhrystone		embench picojpeg
		embench qrduino
		embench sglib-combined
		embench slre
		embench st
		embench wikisort
		Coremark

A complete categorization list can be found in table 7.1. So far we do not know if this change of behavior over time influences similarity at all and further study would be required to prove or disprove the necessity of in-flight data. It is additionally important to note that the boundary between group 1 and 3 is rather arbitrary and inherently connected to `NLYNX_SECTION_SIZE` defined in Chapter 4. This parameter acts as a smoothing exposure window, as all behavioral changes within its borders are inherently ignored. Expanding the exposure window will inevitably push workloads from group 3 into group 1, whereas shrinking the window will potentially cause the opposite effect.

7.5 Metric correlation

We calculated the Pearson's correlation coefficient of the different metrics collected, in order to gauge metric interdependence. A coefficient closer to 0 indicates lower correlation, which is desired. A metric is intended to represent a single subsystem in the core and each subsystem is meant to be represented by a single metric. More correlation means more bias towards one particular subsystem in the data. For this calculation, we excluded benchmarks `factorial`, `nqueens`, `tak`, and `uart_test` due to their very short length. Table 7.2 shows the correlation coefficient between metrics. We see moderate correlation with the mean being about 0.25.

There are outliers however. We see very strong correlation between the fetch coefficient and the cycles wasted. This is to be expected, as more fetches require more accesses to higher level memory systems which results in more wasted cycles. We also found strong correlation between the share of branch instructions and the fetch coefficient. While this isn't surprising given most workloads seem to favor *branches taken* (as suppose to *branches not taken* implemented by RI5CY), more branches directly translates into more

Table 7.2: Pearson's correlation coefficient between metrics

	%Branch	% br. taken	%FPU	%Jumps	mean dist. HWL	mean it. HWL	fetch coeff.	cycles wasted / instruction
% Mult	−0.37	0.24	−0.09	−0.27	0.26	0.26	−0.04	−0.19
% Branch		−0.12	0.02	0.39	−0.10	−0.11	0.79	0.72
% br. taken			−0.10	−0.20	0.36	0.36	−0.11	−0.03
% FPU				−0.01	−0.03	−0.03	−0.05	0.13
% Jumps					−0.19	−0.18	0.37	0.45
mean dist. HWL						0.06	−0.19	−0.17
mean it. HWL							−0.20	−0.18
fetch coeff.								0.82

instructions fetched. What was surprising was the rather small correlation between the fetch coefficient and the share of taken branches.

Finally, there appeared to be strong correlation between HWL initialization distance and mean HWL iteration count (correlation coefficient of 1.0). This would mean workloads which use HWLs more often seem to strongly prefer less iterations, while workloads with more distance between HWLs favor more iterations. This is a misleading conclusion however, as this effect is solely caused by `embench crc32`, a workload already mentioned for its extensive use of HWLs in Section 7.2. Removing this data point from the set drops the correlation coefficient to 0.06. As mentioned in Section 7.2, `embench crc32` likely presents an extreme edge case, where high iteration counts of HWLs inflate the mean distance between HWL initializations.

CHAPTER 8

Future Work and Conclusions

Future work might develop a method of selecting a subset of benchmarks for a given workload. This would allow us to simply run this reduced set and garner the metrics needed to accurately predict target workload performance instead of blindly running benchmarks, thus judging overall system performance when only certain aspects might be needed. We plan to run a real world workload against our modified system to test this hypothesis. This tool would need to calculate weighting coefficients to correctly estimate the performance of the target workload. Given the overlap of the metrics proposed in this paper, this future effort might choose to remove one or more metrics from its suite.

Additionally, further work might be interested in adapting the metrics collected to a more general processor model. The metrics implemented in this thesis have been simplified vastly given the minimalist nature of the RI5CY memory system. For example, expanding to include metrics such as branch transition rate [HSF00] instead of just branch direction would benefit similarity measurement on platforms employing more complex branch prediction schemes. A further look into the relevance of temporal changes behavior might also be of interest. If these changes prove to be relevant beyond a certain degree, a scheme akin to [JEBJ08] might be of more use than simply measuring static metrics, as complexity for finding stand-in workloads would increase even further. These possible expansions were outside of the scope of this thesis but would be imperative to rule out implementation-reliant metrics and other influences.

8.1 Conclusions

We have presented Datalynx, a new way to gather embedded workload performance telemetry using a feature-aware approach to program similarity. We demonstrated the capabilities of this framework using industry-standard benchmarks and have thus laid the groundwork for more optimized implementations. While our naive implementation is rather hardware resource-intensive, we have been able to provide a proof of concept

that may be modified to make sense outside of academic applications. Datalynx was implemented as an add-on to the open-source RI5CY core. We were able to gather performance data from the benchmarks validated on and demonstrate the logging capabilities of the system. We additionally calculated the correlation coefficient between the metrics collected, which showed high correlation between some of them. At the same time we were also able to show that this correlation does not show the whole image as counterexamples could already be shown in our rather limited data. We saw very strong correlation between the mean HWL initialization distance and mean HWL iteration count but were not able to completely rule out independence of the variables either.

Lastly, we have provided performance profiles for the applications tested on Datalynx.

List of Figures

List of Tables

List of Abbreviations

AbOp abstract operation

ADDMUL addition/multiplication

ALU arithmetic logic unit

API application programming interface

APU auxiliary processing unit

BRAM block RAM

BUFG global clock buffer

COMP comparison/bit manipulation

CSR control and status register

CONV conversion among FP formats/to/from integers

DIVSQRT division/square root

DSP digital signal processing

EDP energy-delay product

FF flip flop

FPU floating point unit

HWL hardware loop

ILP instruction level parallelism

IO input-output

IP intellectual property

ISA instruction set architecture

LUT look-up table

LUTRAM LUT RAM

MCU microcontroller unit

MMCM mixed-mode clock manager

MPU memory protection unit

PC performance counter

PCA principal component analysis

PL programmable logic

PS programmable software

PS7 processing system of the Zynq-7000

RAM random access memory

SIMD Single Instruction, Multiple Data

SoC System On Chip

SPEC Standard Performance Evaluation Corporation

Bibliography

[BE13] Maximilien B. Breughe and Lieven Eeckhout. Selecting representative benchmark inputs for exploring microprocessor design spaces. *ACM Transactions on Architecture and Code Optimization*, 10(4):37:1–37:24, December 2013.

[Ben21] Benchmark (computing). https://en.wikipedia.org/wiki/ Benchmark_(computing), March 2021.

[CBNV13] Rosario Cammarota, Laleh Aghababaie Beni, Alexandru Nicolau, and Alexander V. Veidenbaum. Optimizing Program Performance via Similarity, Using a Feature-Agnostic Approach. In Chenggang Wu and Albert Cohen, editors, *Advanced Parallel Processing Technologies*, Lecture Notes in Computer Science, pages 199–213, Berlin, Heidelberg, 2013. Springer.

[DD98] Jozo J. Dujmovic and Ivo Dujmovic. Evolution and evaluation of SPEC benchmarks. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):2–9, December 1998.

[Emb] Embench: A Modern Embedded Benchmark Suite. https://www.embench.org/.

[EV03] Lieven Eeckhout and Hans Vandierendonck. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, page 33, 2003.

[EVB03] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing computer architecture research workloads. *Computer*, 36(2):65–71, February 2003.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*, Toulouse, France, 2016.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis*

(WCET 2010), volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Git21] GitHub, pulp-platform/pulpissimo. https://github.com/pulp-platform/pulpissimo, April 2021.

[GL12] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[GRE+01] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, December 2001.

[GST+17] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, October 2017.

[His22] History of RISC-V. https://riscv.org/about/history/, March 2022.

[HSF00] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 241–250, January 2000.

[JEBJ08] Ajay Joshi, Lieven Eeckhout, Robert H. Bell, and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization*, 5(2):10:1–10:33, September 2008.

[JPEJ06] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L.K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, June 2006.

[KHC+13] Dongjin Kim, Yongman Han, Seong-je Cho, Haeyoung Yoo, Jinwoon Woo, Yunmook Nah, Minkyu Park, and Lawrence Chung. Measuring similarity of windows applications using static and dynamic birthmarks. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1628–1633, New York, NY, USA, March 2013. Association for Computing Machinery.

[KHH07] C. Kao, S. Huang, and I. Huang. A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(3):530–543, March 2007.

[MGDP18] Niccolò Marastoni, Roberto Giacobazzi, and Mila Dalla Preda. A deep learning approach to program similarity. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, MASES 2018, pages 26–35, New York, NY, USA, September 2018. Association for Computing Machinery.

[MSM+18] A. Cristiano I. Malossi, Michael Schaffner, Anca Molnos, Luca Gammaitoni, Giuseppe Tagliavini, Andrew Emerson, Andrés Tomás, Dimitrios S. Nikolopoulos, Eric Flamand, and Norbert Wehn. The transprecision computing paradigm: Concept, design, and applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1105–1110, March 2018.

[MSZB21] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(4):774–787, April 2021.

[NRM20] Aravind Nair, Avijit Roy, and Karl Meinke. funcGNN: A Graph Neural Network Approach to Program Similarity. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, pages 1–11, New York, NY, USA, October 2020. Association for Computing Machinery.

[Pat19] David Patterson. Embench™: Recruiting for the Long Overdue and Deserved Demise of Dhrystone as a Benchmark for Embedded Computing. https://www.sigarch.org/embench-recruiting-for-the-long-overdue-and-deserved-demise-of-dhrystone-as-a-benchmark-for-embedded-computing/, June 2019.

[PHB13] James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *arXiv:1308.5174 [cs]*, September 2013.

[PJEJ05] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 10–20, March 2005.

[PRL+19] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, July 2019.

[RIS22] RISC-V Instruction Set Manual. RISC-V, February 2022.

[SPC01] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[SRP+18] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: An Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, October 2018.

[SS96] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[VB04] Hans Vandierendonck and Koen De Bosschere. Many Benchmarks Stress the Same Bottlenecks. *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, page 9, 2004.

[VEE10] Luk Van Ertvelde and Lieven Eeckhout. Benchmark synthesis for architecture and compiler exploration. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11, December 2010.

[VSG+20] Jure Vreča, Karl J. X. Sturm, Ernest Gungl, Farhad Merchant, Paolo Bientinesi, Rainer Leupers, and Zmago Brezočnik. Accelerating Deep Learning Inference in Constrained Embedded Devices Using Hardware Loops and a Dot Product Unit. *IEEE Access*, 8:165913–165926, 2020.

[ZIV94] V. ZIVOJNOVIC. DSPstone : A DSP-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994*, pages 715–720, 1994.

[Zyn] Zynq-7000 SoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

APPENDIX A

How To

In order to build Datalynx for yourself, clone the modified Pulpissimo repository. This was tested on Ubuntu 16.04 with Xilinx Vivado 2018.3. Make sure to complete all steps for building Pulpissimo as described in the top level Readme file. After the `update-ips` script has been executed, navigate to `fpga/pulpissimo-zedboard` and execute the `patch-ips.sh` script to apply the necessary patches to the IPs provided by Pulpissimo. This will add the modifications for your design to contain Datalynx. Do not do this if you intend to build for platforms other than Zedboard. No other hardware is supported at this time! After that you can simulate the design and compile a bitstream as normally.

After bitstream generation, navigate to `fpga/pulpissimo-zedboard/sw` and call `make` to build the embedded Linux image containing the modified Pulpissimo design. If Xilinx Bootgen returns an error upon first stage bootloader generation, call make again from within Xilinx `xsct`. After finishing the make process, the final files can be flashed onto an SD-card using the `install.sh` script. Be careful as this deletes all contents preset on the selected medium! On Ubuntu 16.04 it is possible to encounter issues when trying to boot from the generated SD-card. The simplest solution is copying the final files and flash script to a newer version of Ubuntu.

In order to activate the Datalynx extension during runtime, modify your workload to change the value of the register address `0x7A2` first to `0b110` to reset the counters and then to `0b001` to start counting. After the workload has finished, set the register to `0b010` to signal completion to Linux.

To log data on Linux, open a remote session to the ARM-machine, either by serial console or SSH. Call `./delynx <workload>` from the `/root/` folder prior to program execution. As soon as Pulpissimo signals start of workload execution by setting `0x7A2` to `0b001`, `delynx` will start logging data into the file `<workload>.csv`. CSV column layout can be seen in Table A.1. The second to last row in the file contains the total

Table A.1: Column layout for CSV files generated by `delynx`; top to bottom is left to right

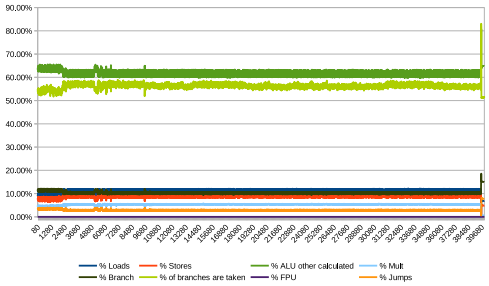
Datapoint	**Description**
Time	Time passed since start of workload execution (in ms)
Instructions	Total count of instructions executed (equal to $2^{\texttt{NLYNX\_SECTION\_SIZE}}$ in in-flight data)
Loads	Number of load instructions encountered
Stores	Number of store instructions encountered
ALU other	Cycles ALU has been active and no other instructions have been executed. Has been replaced by calculating the remainder of unassigned instructions due to the RI5CY ALU being active even when other instructions are executed.
Multiplications	Number of multiplication instructions encountered
Branch	Number of branch instructions encountered
Branches taken	Number of taken branches
FPU	Number of FPU instructions encountered
Jump	Number of jump instructions encountered
HWL init	Number of HWL initializations encountered
HWL jump	Number of HWL jumps performed
Instruction fetch	Number of instructions fetched
Cycles wasted	Number of cycles wasted. This counter is not increased when clock gating is active!

count of each datapoint while the last row indicates the final state of the overflow vector. Should this line contains anything other than a 0, data is to be treated as invalid.

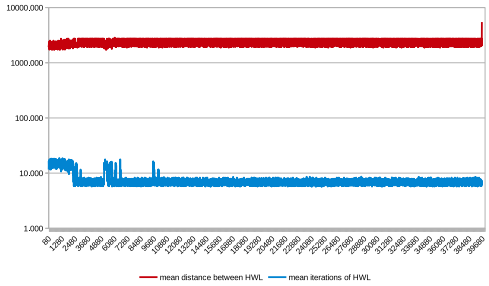
APPENDIX B

Temporal analysis graphs of all measured applications

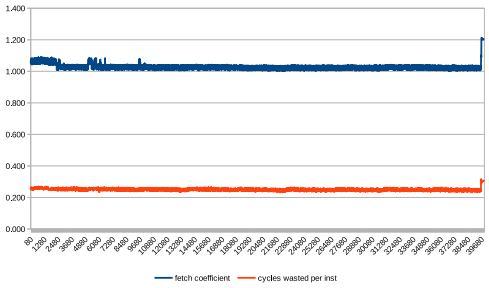
B.1 MiBench



(a) Instruction behavior over time (ms)

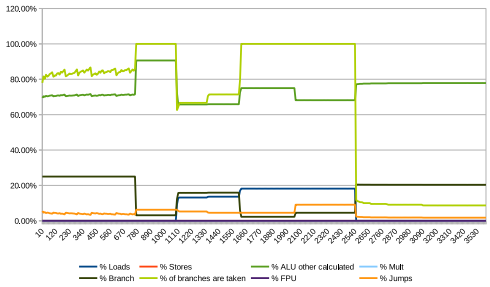


(b) HWL behavior over time (ms)

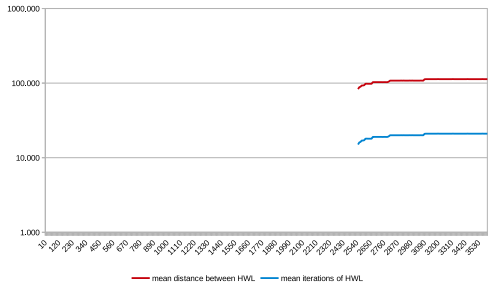


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

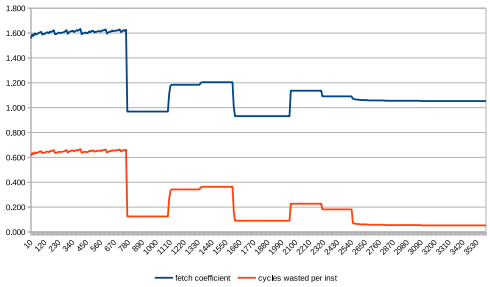
Figure B.1: Performance over time: `basicmath`



(a) Instruction behavior over time (ms)

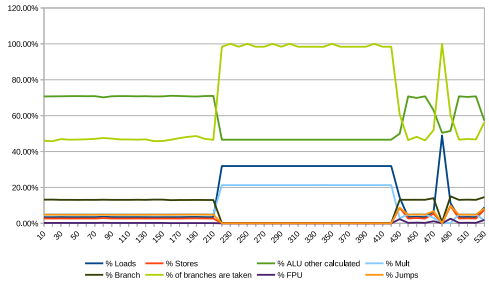


(b) HWL behavior over time (ms)

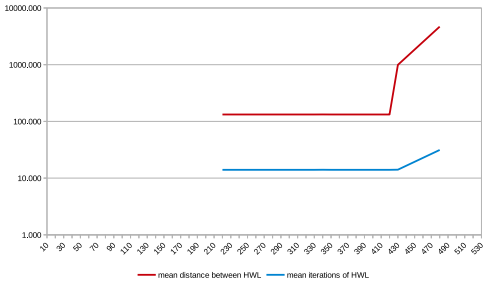


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

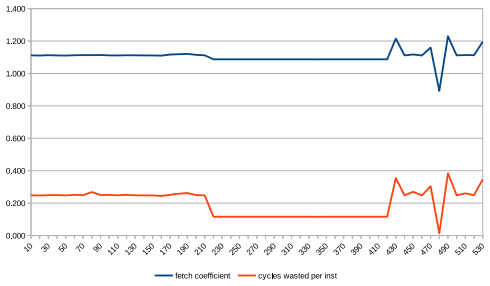
Figure B.2: Performance over time: `bitcount`



(a) Instruction behavior over time (ms)

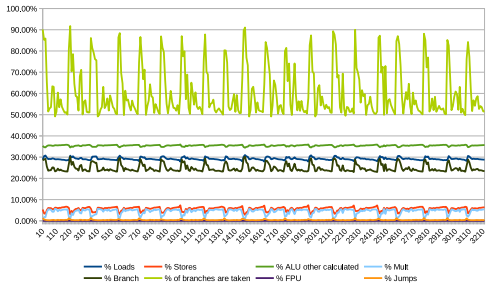


(b) HWL behavior over time (ms)

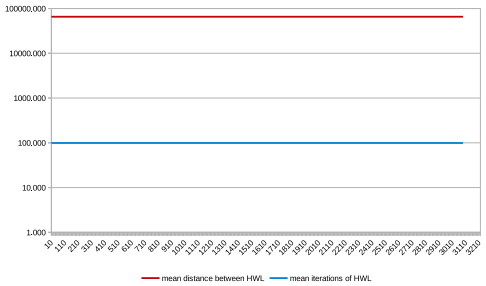


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

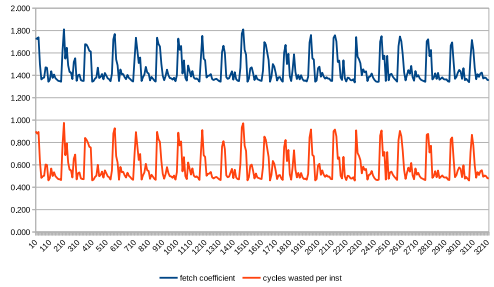
Figure B.3: Performance over time: `susan`



(a) Instruction behavior over time (ms)

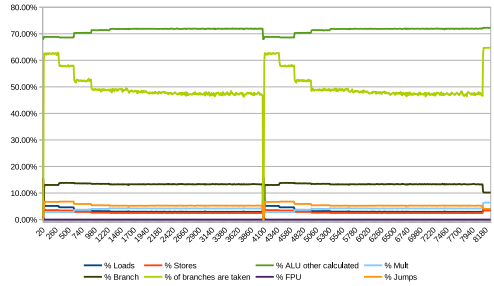


(b) HWL behavior over time (ms)

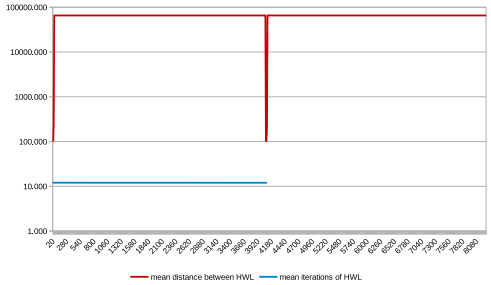


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

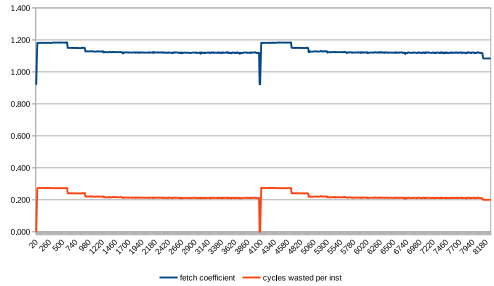
Figure B.4: Performance over time: `dijkstra`



(a) Instruction behavior over time (ms)



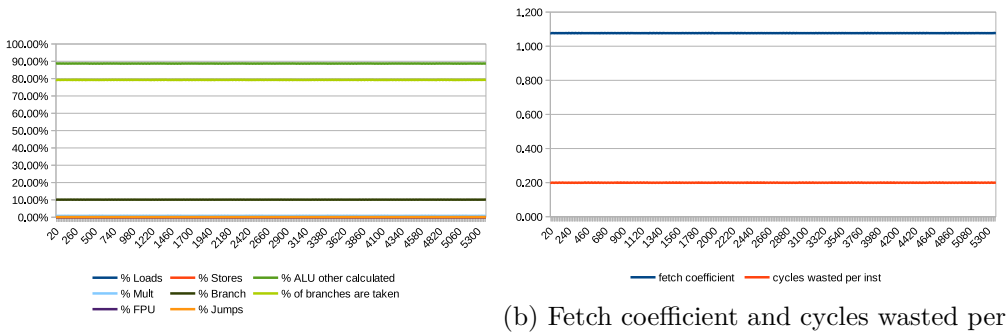
(b) HWL behavior over time (ms)



(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.5: Performance over time: `fft`

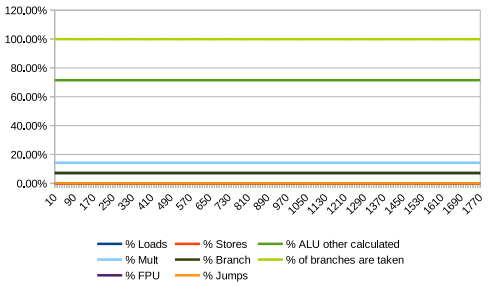
B.2 EmBench



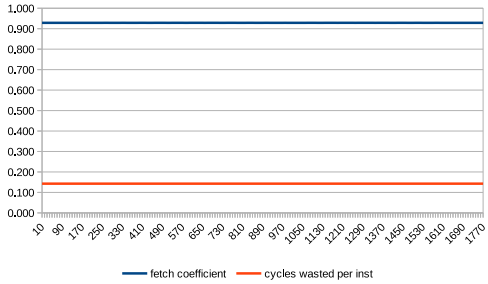
(a) Instruction behavior over time (ms)

(b) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.6: Performance over time: `aha-mont64`

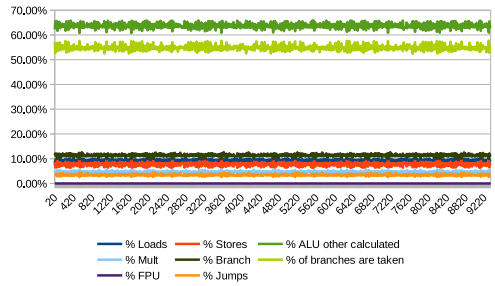


(a) Instruction behavior over time (ms)

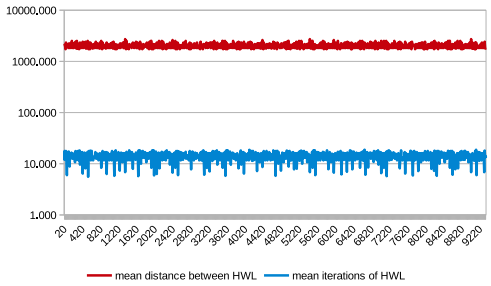


(b) Fetch coefficient and cycles wasted per instruction over time (ms)

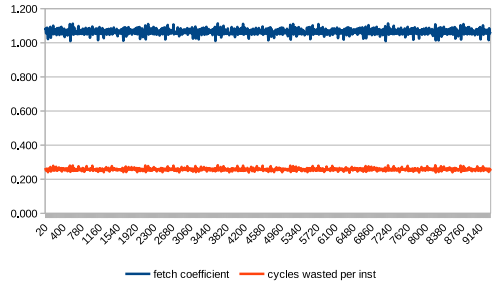
Figure B.7: Performance over time: `crc32`



(a) Instruction behavior over time (ms)

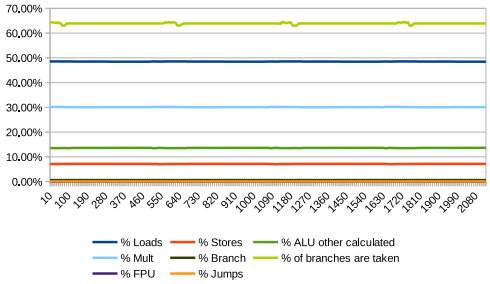


(b) HWL behavior over time (ms)

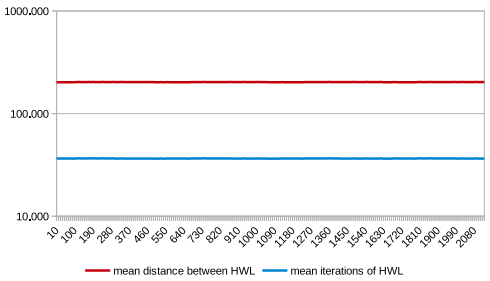


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

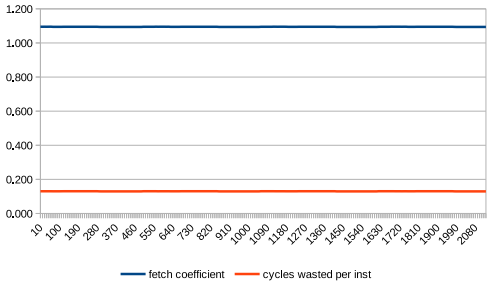
Figure B.8: Performance over time: `cubic`



(a) Instruction behavior over time (ms)

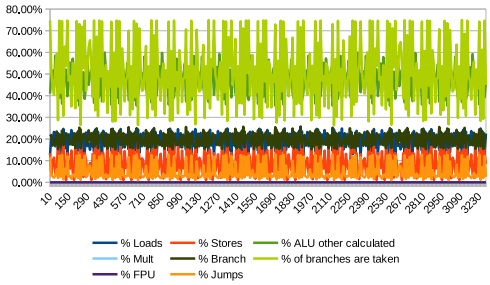


(b) HWL behavior over time (ms)

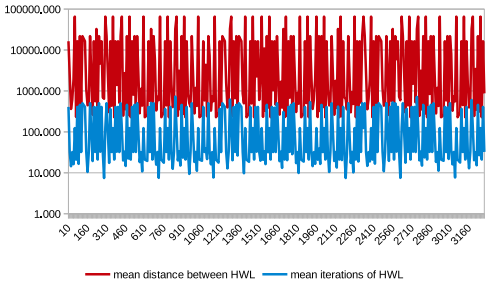


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

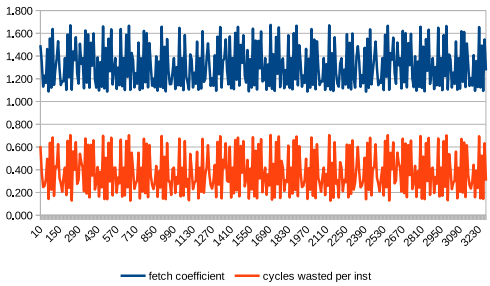
Figure B.9: Performance over time: `edn`



(a) Instruction behavior over time (ms)

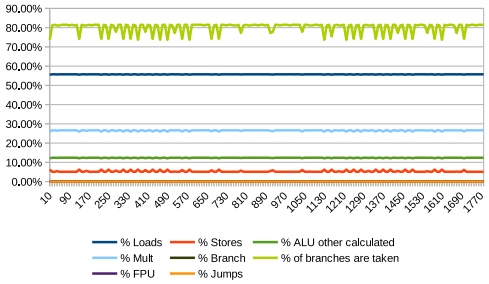


(b) HWL behavior over time (ms)

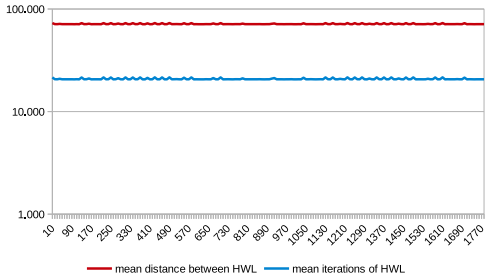


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

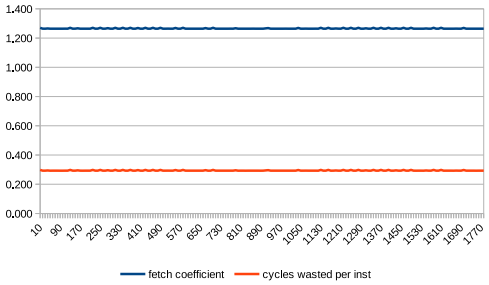
Figure B.10: Performance over time: `huffbench`



(a) Instruction behavior over time (ms)

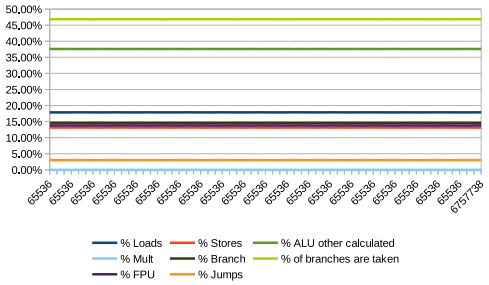


(b) HWL behavior over time (ms)



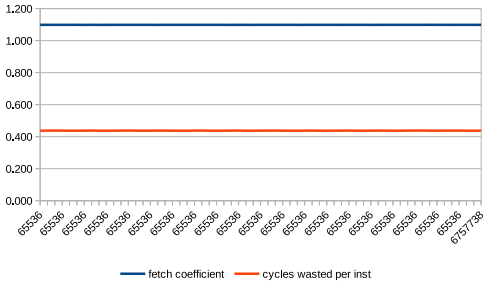
(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.11: Performance over time: `matmult-int`



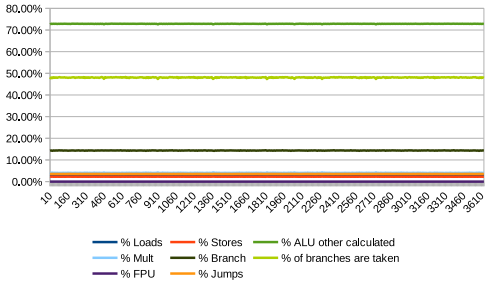
(a) Instruction behavior over time (ms)

(b) HWL behavior over time (ms)

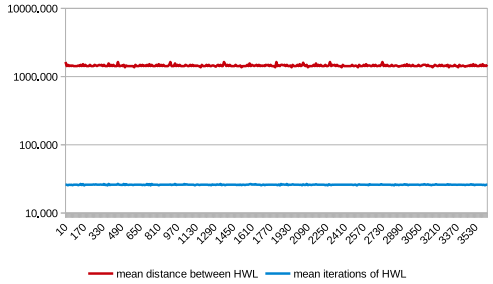


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

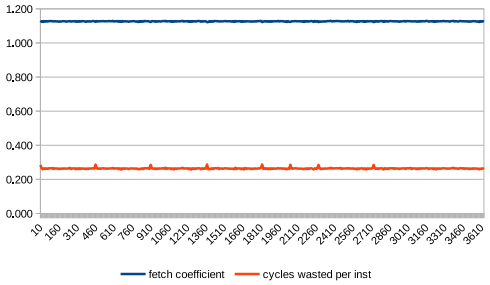
Figure B.12: Performance over time: `minver`



(a) Instruction behavior over time (ms)

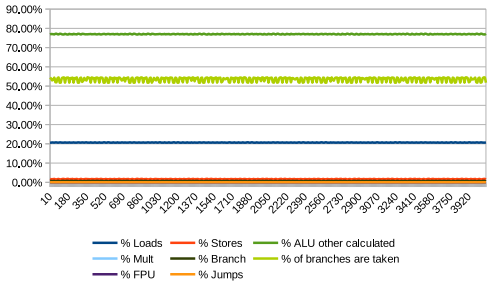


(b) HWL behavior over time (ms)

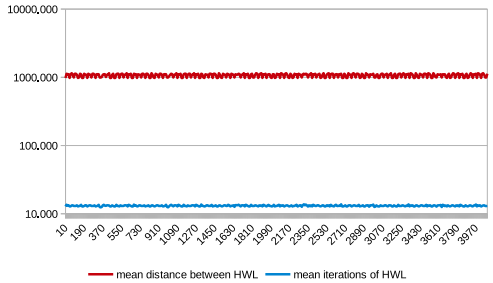


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

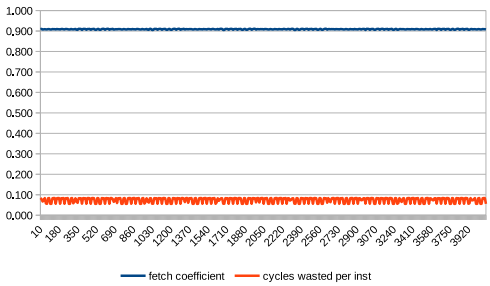
Figure B.13: Performance over time: `nbody`



(a) Instruction behavior over time (ms)

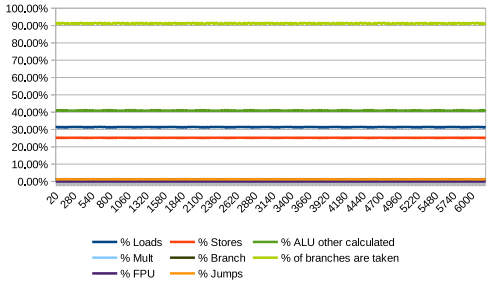


(b) HWL behavior over time (ms)

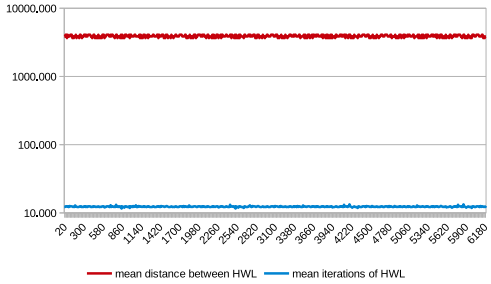


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

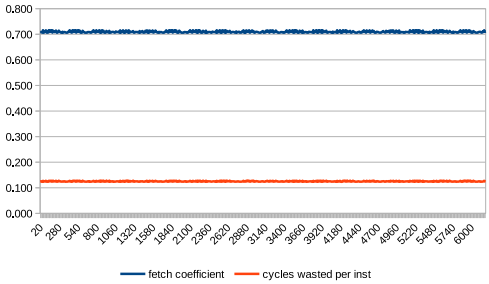
Figure B.14: Performance over time: `nettle-aes`



(a) Instruction behavior over time (ms)

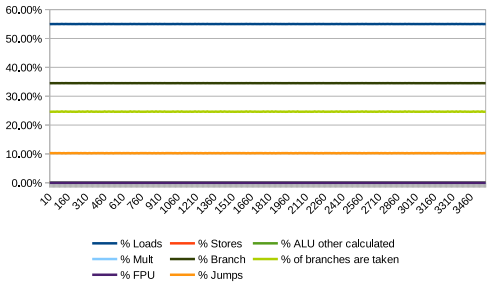


(b) HWL behavior over time (ms)

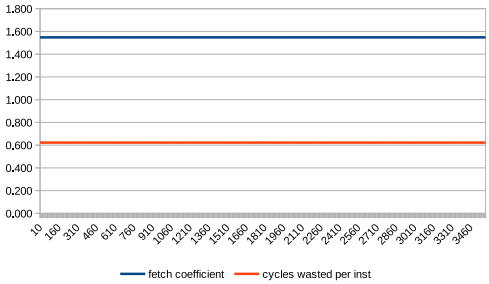


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.15: Performance over time: `nettle-sha256`

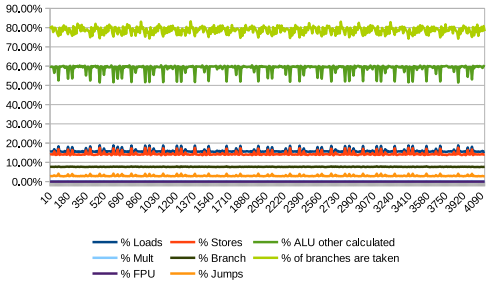


(a) Instruction behavior over time (ms)

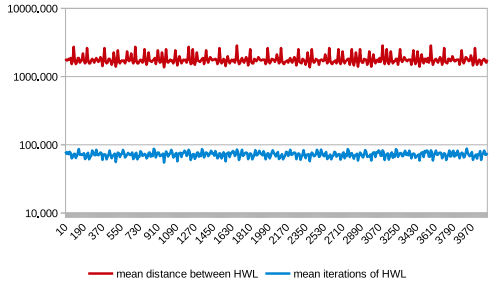


(b) Fetch coefficient and cycles wasted per instruction over time (ms)

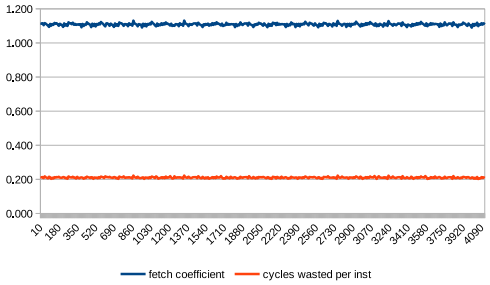
Figure B.16: Performance over time: `nsichneu`



(a) Instruction behavior over time (ms)

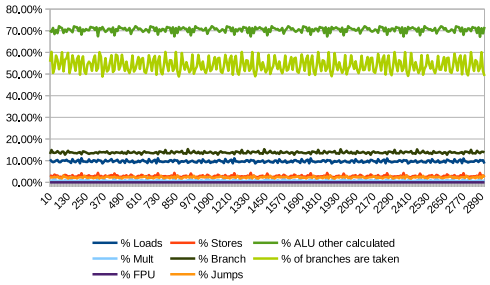


(b) HWL behavior over time (ms)

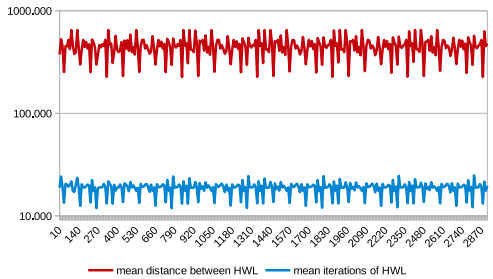


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

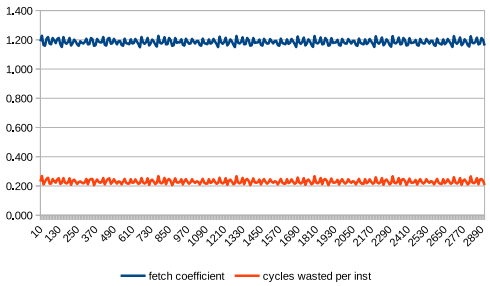
Figure B.17: Performance over time: `picojpeg`



(a) Instruction behavior over time (ms)

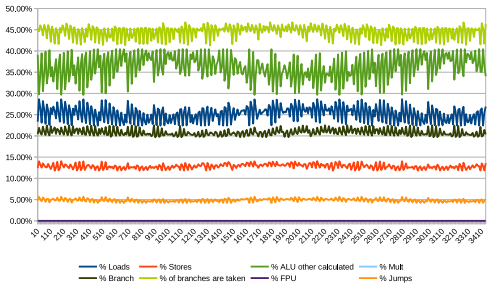


(b) HWL behavior over time (ms)

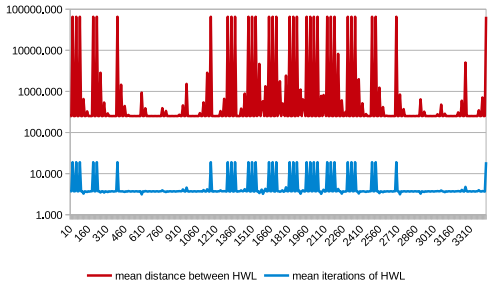


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

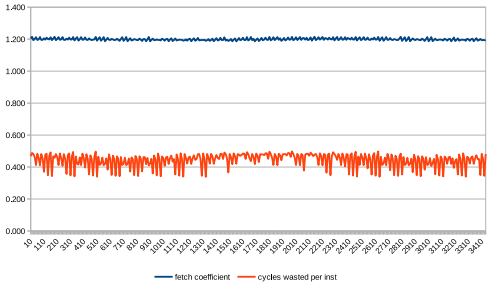
Figure B.18: Performance over time: `qrduino`



(a) Instruction behavior over time (ms)

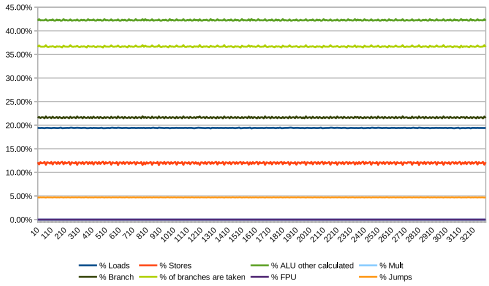


(b) HWL behavior over time (ms)

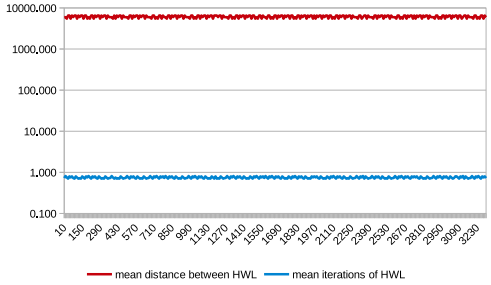


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

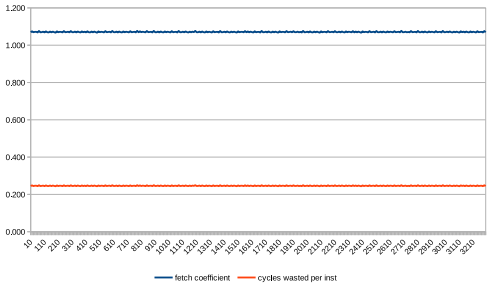
Figure B.19: Performance over time: `sglib-combined`



(a) Instruction behavior over time (ms)

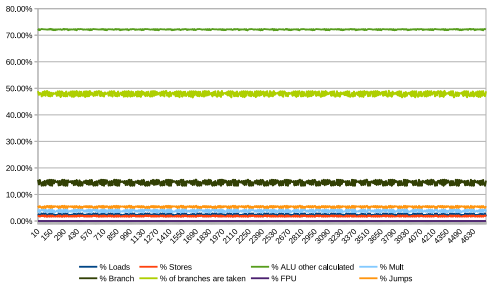


(b) HWL behavior over time (ms)

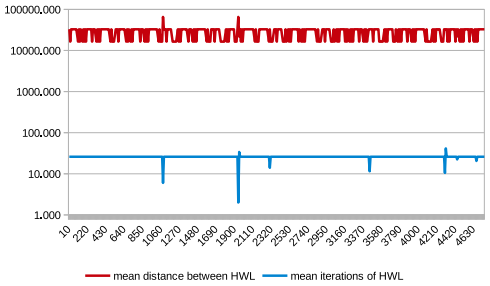


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

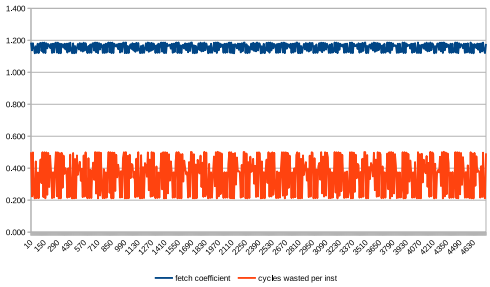
Figure B.20: Performance over time: `slre`



(a) Instruction behavior over time (ms)

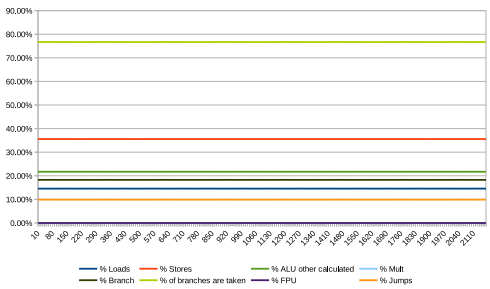


(b) HWL behavior over time (ms)

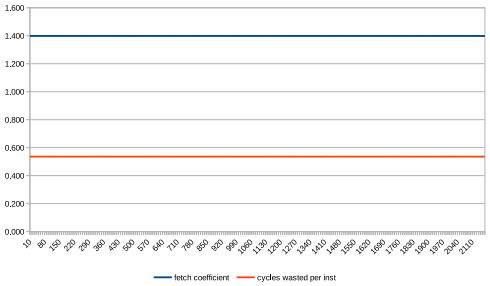


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.21: Performance over time: `st`

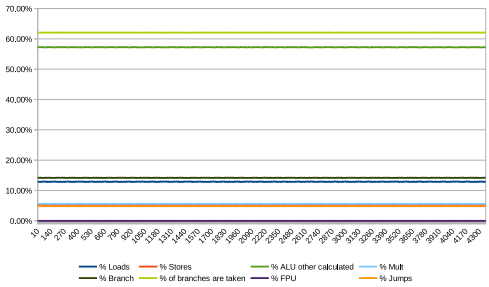


(a) Instruction behavior over time (ms)

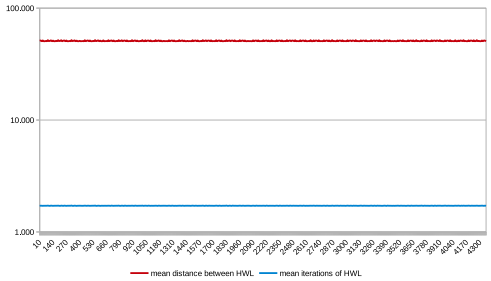


(b) Fetch coefficient and cycles wasted per instruction over time (ms)

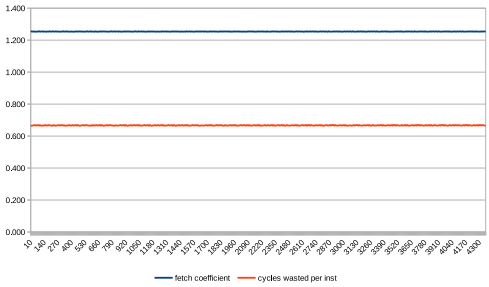
Figure B.22: Performance over time: `statemate`



(a) Instruction behavior over time (ms)

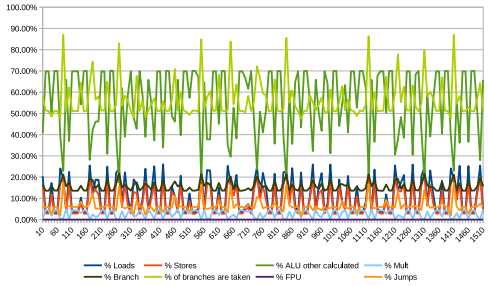


(b) HWL behavior over time (ms)

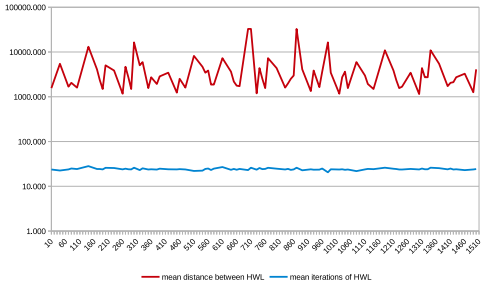


(c) Fetch coefficient and cycles wasted per instruction over time (ms)

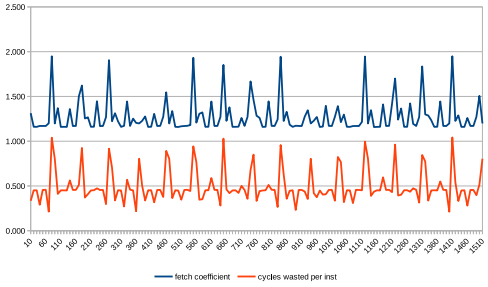
Figure B.23: Performance over time: `ud`



(a) Instruction behavior over time (ms)



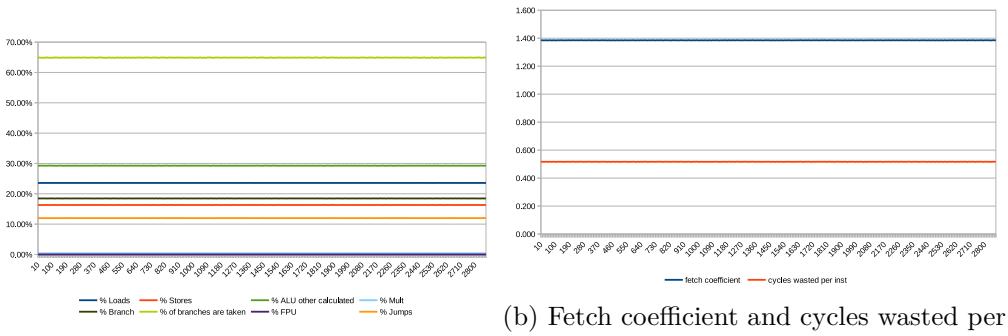
(b) HWL behavior over time (ms)



(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.24: Performance over time: `wikisort`

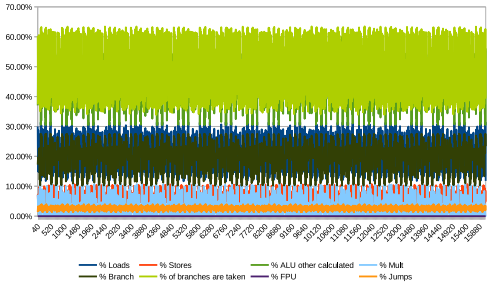
B.3 Other



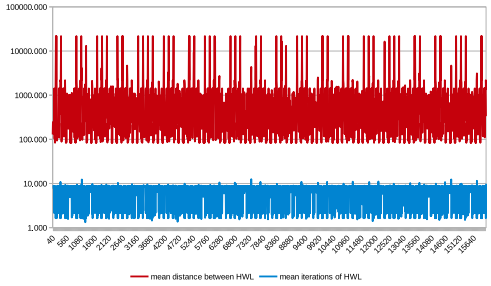
(a) Instruction behavior over time (ms)

(b) Fetch coefficient and cycles wasted per instruction over time (ms)

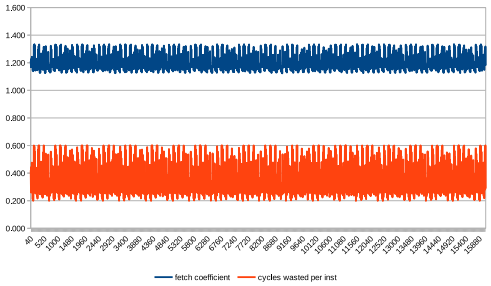
Figure B.25: Performance over time: Dhrystone



(a) Instruction behavior over time (ms)



(b) HWL behavior over time (ms)



(c) Fetch coefficient and cycles wasted per instruction over time (ms)

Figure B.26: Performance over time: Coremark