



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Enabling high performance dynamic language programming for micro-core architectures

Maurice Jamieson



Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2022

Abstract

Micro-core architectures are intended to deliver high performance at a low overall power consumption by combining many simple central processing unit (CPU) cores, with an associated small amount of memory, onto a single chip. This technology is not only of great interest for embedded, Edge and IoT applications but also for High-Performance Computing (HPC) accelerators. However, micro-core architectures are difficult to program and exploit, not only because each technology is different, with its own idiosyncrasies, but also because they each present a different low-level interface to the programmer. Furthermore, micro-cores have very constrained amounts of on-chip, scratchpad memory (often around 32KB), further hampering programmer productivity by requiring the programmer to manually manage the regular loading and unloading of data from the host to the device during program execution. To help address these issues, dynamic languages such as Python have been ported to several micro-core architectures but these are often delivered as interpreters with the associated performance penalty over natively compiled languages, such as C.

The research questions for this thesis target four areas of concern for dynamic programming languages on micro-core architectures: (RQ1) how to manage the limited on-chip memory for data, (RQ2) how to manage the limited on-chip memory for code, (RQ3) how to address the low runtime performance of virtual machines and (RQ4) how to manage the idiosyncratic architectures of micro-core architectures. The focus of this work is to address these concerns whilst maintaining the programmer productivity benefits of dynamic programming languages, using ePython as the research vehicle. Therefore, key areas of design (such as abstractions for offload) and implementation (novel compiler and runtime techniques for these technologies) are considered, resulting in a number of approaches that are not only applicable to the compilation of Python codes but also more generally to other dynamic languages on micro-cores architectures.

RQ1 was addressed by providing support for kernels with arbitrary data size through high-level programming abstractions that enable access to the memory hierarchies of micro-core devices, allowing the deployment of real-world applications, such as a machine learning code to detect cancer cells in full-sized scan images. A new abstract machine, Olympus, addressed RQ2 by supporting the compilation of dynamic languages, such as Python, to micro-core native code. Olympus enables ePython to close the kernel runtime performance gap with native C, matching C for the LINPACK and an iterative Fibonacci benchmark, and to provide, on average, around 75% of native C

runtime performance across four benchmarks running on a set of eight CPU architectures. Olympus also addresses RQ3 by providing dynamic function loading, supporting kernel codes larger than the on-chip memory, whilst still retaining the runtime performance benefits of native code generation. Finally, RQ4 was addressed by the Eithne benchmarking framework which not only enabled a single benchmarking code to be deployed, unchanged, across different CPU architectures, but also provided the underlying communications framework for Olympus. The portability of end-user ePython codes and the underlying Olympus abstract machine were validated by running a set of four benchmarks on eight different CPU architectures, from a single codebase.

Acknowledgements

I would especially like to thank my supervisor, Dr. Nick Brown, for all his advice, guidance and encouragement throughout the project. I have been very lucky to have Nick as my supervisor, without whose support I doubt I would have been able to complete my research due to the impact of the pandemic. In addition, Nick's ePython provided a solid foundation for my own research.

I would also like to thank Adrian Jackson and Dr. Mark Bull for their helpful challenges, suggestions and encouragement, especially when I was lost in the depths of implementation detail.

I would like to thank my examiners, Dr. Michèle Weiland and Prof. Phil Trinder, for all their insightful comments, corrections and suggestions that have helped to improve this thesis.

I would like to thank everyone at the EPCC for making me feel at home and especially to Ben Morse for all his help and guidance.

A special mention to Steven James for the conversations over many years that led me down this path. I have learned so much through my research and only now can I understand what Steve's been saying to me over the last 30 years.

Finally, I would like to thank my family for all their support throughout the project and a very special thank you to my partner and best friend, Jane, without whom I would not have been able to start, let alone complete, this endeavour.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Maurice Jamieson)

In memory of my father,
Maurice William Jamieson

Table of Contents

1	Introduction	1
1.1	Motivation	2
1.2	Research questions	3
1.3	Research software outputs	4
1.4	Contributions	5
1.5	Publications	6
1.6	Thesis structure	7
2	Background and related work	9
2.1	Overview	10
2.2	Micro-core processors	10
2.2.1	Micro-core architecture	14
2.3	Graphics Processing Units (GPUs)	15
2.4	Field-programmable gate arrays (FPGAs)	18
2.5	ePython	19
2.6	Accelerating Python codes	23
3	Design of enhancements to ePython	29
3.1	Overview	30
3.2	Managing arbitrary data size (RQ1)	31
3.2.1	Domain Specific Language for micro-cores	31
3.2.2	Host / device memory model	32
3.2.2.1	Pass by reference	33
3.2.2.2	Memory kinds	35
3.3	Managing arbitrary code size (RQ2)	37
3.3.1	Dynamic code loading	37

3.4	Increasing kernel runtime performance (RQ3)	38
3.4.1	ePython object typing	38
3.4.1.1	Type inferencing	39
3.4.1.2	Dynamic typing and polymorphism	40
4	Implementation of enhancements to ePython	44
4.1	Overview	45
4.2	Managing arbitrary data size (RQ1)	46
4.2.1	Communication model	46
4.2.2	Host Python integration	47
4.2.2.1	CPython integration limitations	48
4.2.3	Communication protocol	49
4.2.4	ePython communication primitives	52
4.2.5	Memory hierarchies and memkinds	53
4.3	Increasing kernel runtime performance (RQ3)	55
4.3.1	Native code generation options	55
4.3.2	The Merlin virtual machine	56
4.3.3	Olympus compiler and abstract machine	59
4.3.3.1	Overview	60
4.3.3.2	Memory model	63
4.3.3.3	Type management	65
4.3.3.4	Memory management	65
4.3.3.5	Function stack frame management	67
4.3.3.6	Code management	70
4.4	Managing arbitrary code size (RQ2)	72
4.4.1	Olympus dynamic function loading	73
4.4.2	Implementing the dynamic loader support	74
4.4.3	Summary	76
5	Experimental Environment	78
5.1	Overview	79
5.2	Additional micro-core architectures	80
5.2.1	Cerberus soft micro-core processor	81
5.2.1.1	RISC-V soft-core selection	84
5.2.2	Eithne micro-core benchmarking framework	85
5.2.2.1	Architecture overview	85

5.2.2.2	Host and device communications flow	87
5.2.2.3	Eithne API kernel modifications	88
5.2.2.4	Host driver code	89
5.2.3	Eithne initialisation and data transfer timings	90
5.3	Test environment	93
5.3.1	CPU selection	93
5.3.2	C compiler selection	94
5.3.3	Benchmark selection	94
5.3.3.1	Machine learning (ML) image processing	94
5.3.3.2	Fibonacci sequence	95
5.3.3.3	Jacobi iteration	95
5.3.3.4	LINPACK	96
5.3.3.5	Sieve of Eratosthenes (Byte Sieve)	96
5.3.4	Benchmark rationale and approach	97
6	Results	99
6.1	Overview	100
6.2	Managing arbitrary data size (RQ1)	100
6.2.1	ML image processing	100
6.3	Increasing kernel runtime performance (RQ3)	105
6.3.1	Merlin virtual machine	105
6.3.1.1	Fibonacci benchmark: kernel code size and runtime performance	105
6.3.2	Olympus abstract machine	108
6.3.2.1	Fibonacci benchmark: kernel code size	109
6.3.2.2	Fibonacci benchmark: runtime performance	110
6.3.2.3	Jacobi benchmark: kernel code size	113
6.3.2.4	Jacobi benchmark: kernel runtime performance	114
6.3.2.5	LINPACK benchmark: kernel code size	116
6.3.2.6	LINPACK benchmark: kernel runtime performance	118
6.3.2.7	Sieve of Eratosthenes benchmark: kernel code size	121
6.3.2.8	Sieve of Eratosthenes benchmark: kernel runtime performance	124
6.4	Managing arbitrary code size (RQ2)	127
6.4.1	Jacobi benchmark: dynamically loaded code size	129

6.4.2	Jacobi benchmark: dynamic loaded code runtime performance	132
6.5	Summative Olympus / native C analysis	133
6.5.1	Summative analysis: kernel code size	133
6.5.2	Summative analysis: kernel runtime performance	136
6.6	Desktop comparison: LINPACK benchmark	139
6.6.1	LINPACK benchmark: code size	140
6.6.2	LINPACK benchmark: runtime performance	141
6.7	Summary	142
7	Conclusions and future work	144
7.1	Conclusions	145
7.2	Summary of contributions	147
7.3	Research impact	148
7.4	Limitations	148
7.4.1	ePython language	148
7.4.2	Type inferencing	149
7.4.3	C integration / Olympus application binary interface	149
7.4.4	Limited micro-core architecture evaluation	149
7.5	Future work	150
7.5.1	Automatic memory management for data and code	150
7.5.2	Optimisation of the Olympus abstract machine	150
7.5.2.1	Mnemonics: size versus runtime performance	150
7.5.2.2	Automatic dynamic function selection	151
7.5.2.3	Data types	151
7.5.2.4	Register allocation	152
7.5.3	Compiler intermediate representation	152
7.5.4	Additional device support: GPUs and FPGAs	153
	Bibliography	154
A	Fibonacci sequence code listings	174
B	LINPACK benchmark code listings	185
C	Sieve of Eratosthenes code listings	245
D	Merlin Engine (VM) code listings	254

E	PicoRV32 support files	256
F	Olympus mnemonics	259

As the cost of powering a supercomputer or a datacenter increases, next generation exascale systems need to be considerably more power- and energy-efficient than current supercomputers to be of practical use.

Performance and Architecture Lab (PAL) at PNNL, “Tackling the Power and Energy Wall for Future HPC Systems” [1]

1

Introduction

The new generation of supercomputers, known as *exascale* computing, are designed to process 10^{18} double precision floating point operations per second (FLOPS) [2]. Whilst this will significantly increase the processing power of exascale machines over the previous generation of supercomputers, the increasing energy consumption of successive generations is approaching a limit known as the *power wall* [1]. The Defense Advanced Research Projects Agency (DARPA) has set a power consumption target of 20 Megawatts (MW) for exascale machines, resulting in an energy efficiency figure of 50 GFLOPS per Watt [3]. In order to achieve this, the use of hybrid High-Performance Computing (HPC) architectures with graphics processing units (GPUs) as accelerators has increased, such as the 4:1 ratio of GPUs to central processing units (CPUs) per node of the new OLCF Frontier exascale supercomputer [4]. Other novel architectures for HPC have been introduced, including innovative *micro-core*¹ processor architectures that consist of many, low energy cores combined with small amounts of memory on a single chip, such as the 256 core Kalray MPPA [5], the 256 core Sunway SW26010 [6], the 1024 core Adapteva Epiphany-V [7] and the 2048 core PEZY-SC2 [8]. These micro-core architectures have the promise of overcoming the power wall due to the high energy efficiency of their designs, for example, the class-leading 70 GFLOPS per Watt of the the 64-core Adapteva Epiphany-IV [7] [9]. Furthermore, a number of these processor architectures have already been proven in supercomputer designs, such as the PEZY-SC2 used in the Shoubu system B, ranked #1 in the June 2019 Green500 list [10], and the Sunway SW26010 used in the Sunway TaihuLight, ranked #3 in June 2019 Top500 [11].

1.1 Motivation

The power efficiency of micro-core architectures is not only of interest for HPC but also for embedded systems and *edge computing* [12], where micro-core architectures such as the Adapteva Epiphany-III [13] and X MOS XCore [14] have been successfully used for commercial applications. For example, the Epiphany-III has been used by Ericsson for Long-Term Evolution (LTE) baseband signal processing [15]. Whilst these architectures provide the high energy efficiency and low overall power consumption levels required for embedded devices, micro-cores are notoriously difficult to program and take advantage of; each technology is different with its own idiosyncrasies, such

¹Although the term *manycore* is commonly used, we define micro-cores as manycores with extremely small amounts of on-chip, scratchpad RAM (circa 32 - 64KB) without hardware cache support.

as the topology of the Network-on-Chip (NOC), and they each present a different low-level interface to the programmer. Although manufacturers have made great progress in developing the hardware, parallel programming and compilation techniques have not evolved quickly enough to exploit this effectively [16]. Fundamentally, writing parallel, scalable code is difficult and requires the programmer to consider multiple levels of parallelism to get good runtime performance [17]. For micro-core architectures, this problem is exacerbated by the design objective to reduce power consumption whilst increasing runtime performance. For example, hardware-controlled caches are replaced by extremely limited ($\leq 64\text{KB}$) on-chip scratchpad Random-Access Memory (RAM) in order to reduce the silicon area by around 34% and provide up to a three times decrease in the per-access energy consumption [18]. However, to date, these technologies have tended to result in significant runtime performance overheads, required the programmer to ensure their code fits within the limited on-chip memory, provided limited choices around data location and size, and provided little, if any, portability across architectures. As evidenced by ePython [19], at the start of this research a Python interpreter for the Epiphany-III, dynamic programming languages can significantly reduce the programming effort required to overcome these complexities in comparison to the provided, low-level C software development kits (SDKs) [20].

1.2 Research questions

Based upon the hypothesis that dynamic programming languages greatly simplify the development of codes for micro-core devices, this research addresses four challenges for dynamic programming languages on these architectures:

- RQ1 How to manage the limited on-chip memory for data?** Many codes, including standard benchmarks such as LINPACK [21], require more data memory than micro-core devices provide.
- RQ2 How to manage the limited on-chip memory for code?** Real-world codes, such as image processing, are larger than the available, extremely limited (c. 32KB), on-chip memory [22].
- RQ3 How to address the low runtime performance of virtual machines?** Whilst virtual machines (VMs) are easier to implement and port to different architectures than compilation to native code [23], they introduce a significant runtime performance overhead.

RQ4 How to manage the idiosyncratic architectures of micro-core architectures?

The unique features of each micro-core architecture, coupled with their own specific drivers and toolset, frustrate the development, testing and portability of both user codes and the underlying runtime support [22].

The focus is to address these questions whilst maintaining the programmer productivity benefits of dynamic programming languages, using ePython as the research vehicle. Therefore, key areas of design (such as abstractions for offload) and implementation (novel compiler and runtime techniques for these technologies) have been considered, resulting in a number of approaches that are not only applicable to the compilation of Python codes but also more generally to other dynamic languages across a range micro-cores architectures.

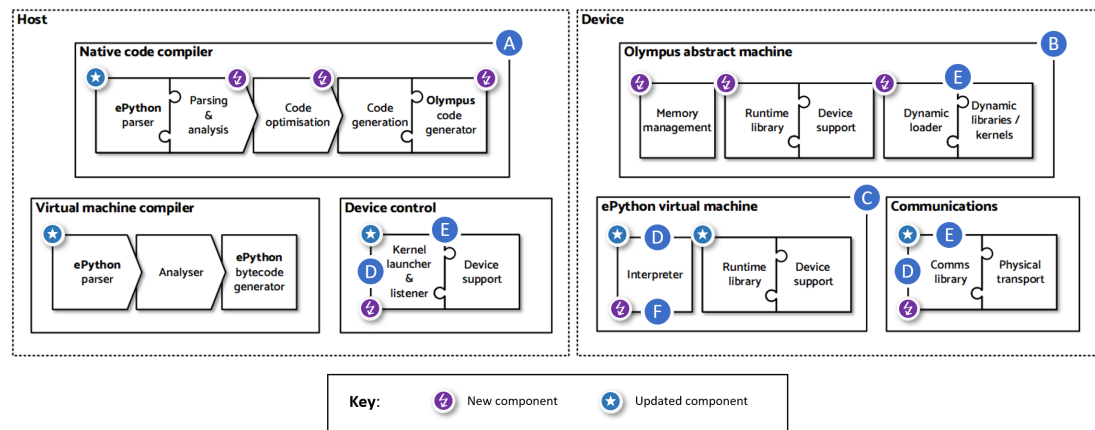
1.3 Research software outputs

Figure 1.1: ePython architecture showing host and device components created and updated by this research.

In order to address RQ1, RQ2 and RQ3, this research added a number of new components to ePython, in addition to updates to the existing components. Figure 1.1 shows the high-level architecture of ePython, grouped by the components running on the host (left-hand side) and on the device (right-hand side), highlighting the components that have been introduced or updated. The key new components are:

- The Olympus native code compiler for ePython, highlighted by **A** in Figure 1.1.

- The Olympus native code abstract machine for dynamic programming languages targeting micro-core architectures (Section 4.3.3) **B**, which replaces the ePython virtual machine running on the device **C**.
- A new communications protocol and library to support a new *pass by reference* model (Section 3.2.2.1) for ePython kernel functions and device *memory hierarchies* (Section 3.2.2.2) **D**.
- A dynamic function loading mechanism for ePython kernels (Section 3.3.1) **E**.
- The Merlin prototype virtual machine (Section 4.3.2), created to support investigations into optimising the bytecode size and runtime performance of the ePython virtual machine **F**.

1.4 Contributions

This thesis provides the following contributions:

- The first *pass by reference* semantics model for data passed as arguments to kernels off-loaded to micro-core accelerators, thereby preserving the programming paradigm of dynamic languages running on traditional CPUs (Section 3.2.2.1) and providing the basis for the *prefetching* of host vector *slices*, addressing RQ1.
- A compact model and implementation (circa 900 bytes) for supporting *memory hierarchies* and *memkinds* [24] within dynamic languages on micro-core accelerators, to enable the deployment of kernels with arbitrary-sized data (Section 3.2.2.2), addressing RQ1.
- The Olympus portable, compact (on average, 1.4 to 2 times bigger than native C), high-performance abstract machine for dynamic languages (Section 4.3.3) that matches ($GM^2 = 101.0\%$) the runtime performance of native C kernels for both the LINPACK and an iterative Fibonacci sequence benchmark, across eight different CPU architectures (Section 6.5.2). Olympus achieves an average (GM) of almost 75% of the runtime performance of native C, across four different benchmarks on all eight CPU architectures (Section 5.3), addressing RQ3.

²Geometric mean.

- A dynamic function loading model for micro-cores (Section 3.3.1) that supports the arbitrary loading and unloading of kernel functions, enabling the deployment of kernels with code size larger than the on-chip memory with an initial memory footprint up to 30% smaller than the equivalent hand-crafted C kernels (Section 6.4), addressing RQ2.
- The Eithne micro-core benchmarking framework [25] that provides a suite of benchmarks that can be run across a number of different devices, with a layered architecture enabling the addition of new C benchmarks with minor modification to the original codes (Section 5.2.2). Eithne abstracts over the idiosyncrasies of different micro-core architectures and provides the underlying communication framework for the Olympus abstract machine, addressing RQ4. Prior to Eithne, there were no benchmarking frameworks for micro-core devices and it remains unique in this regard at the time of submission of this thesis.

1.5 Publications

The following peer-reviewed papers and posters were published during the research of this thesis and form part of the content of Chapters 3, 4, 5 and 6:

- Maurice Jamieson and Nick Brown (in press), “Performance of the Vipera framework for DSLs on micro-core architectures,” *Euro-Par 2022: DSL-HPC workshop*, 2022 [26].
- Maurice Jamieson and Nick Brown, “Compact native code generation for dynamic languages on micro-core architectures,” in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. Association for Computing Machinery, 2021, pp. 131–140 [27].
- Maurice Jamieson, Nick Brown, and Sihang Liu, “Having your cake and eating it: Exploiting Python for programmer productivity and performance on micro-core architectures using ePython,” in *Proceedings of the 19th Python in Science Conference : SciPy 2020*, 2020, pp. 97–105 [20].
- Maurice Jamieson and Nick Brown, “Benchmarking micro-core architectures for detecting disasters at the edge,” in *2020 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, 2020, pp. 27–35 [28].

- Maurice Jamieson and Nicholas Brown, “High level programming abstractions for leveraging hierarchical memories with micro-core architectures,” *Journal of Parallel and Distributed Computing*, vol. 138, pp. 128-138, 2020 [22].

Other publications:

- Maurice Jamieson and Nick Brown, “A Framework for Benchmarking Micro-Core Accelerators”, Poster presented at *Supercomputing 2019 (SC19)*, Denver, Colorado, 2019 [29].
- Nicholas Brown and Maurice Jamieson, “Leveraging hierarchical memories for micro-core architectures”, Extended abstract *5th International Conference on Exascale Applications and Software*, Edinburgh, United Kingdom, 2018 [30].

1.6 Thesis structure

The structure of the thesis is as follows:

- **Chapter 2** describes the background to micro-core architectures in the context of the alternative technologies, specifically GPUs and field-programmable gate arrays (FPGAs), followed by an introduction to ePython and an overview of related work regarding accelerating dynamic programming languages, with particular focus on Python.
- **Chapter 3** details the design considerations for the implementation of dynamic languages on micro-core architectures, with particular regard to micro-core accelerators. Firstly, enhancements to the existing ePython memory model are considered, including memory hierarchy support. Secondly, changes to the language to support native code generation and dynamic code loading. Thirdly, native code generation options, specifically just-in-time (JIT) and ahead-of-time (AOT) compilation. Finally, virtual machine and bytecode modifications to underpin the implementation of a JIT compiler for ePython.
- **Chapter 4** details the implementation of the native code compiler, abstract machine and memory model to enable the support for arbitrary kernel data and code size, allied to increasing runtime performance to that of kernels hand-written in C.

- **Chapter 5** describes the challenges and solutions to providing a range of different micro-core architectures to support the research, including the creation of a novel benchmarking framework (Eithne [28]) to quickly assess new micro-core architectures and a new *soft-core* design (Cerberus [31]) running on FPGAs to provided additional micro-core targets. This is followed by a description of the experimental environment, including the specifications of test devices, compilers and selection of benchmarks.
- **Chapter 6** presents and analyses the benchmark results used to assess the major areas of research that address the challenges for dynamic programming languages on micro-cores architectures: managing arbitrary kernel data size, increasing kernel runtime performance through native code generation and managing arbitrary large kernel code size.
- **Chapter 7** provides a summary of the thesis, highlighting the conclusions, major contributions, research impact and future work.

The top 100 instructions account for all the instructions used by all the processors. This leads us to question why most current processors have more than 100 instructions.

Mutigwe et al., “Instruction Set Usage Analysis for ApplicationSpecific Systems” Design [32]

2

Background and related work

2.1 Overview

This chapter discusses the background and related work to this research. Section 2.2 details the drivers behind micro-core designs and provides an overview of their architecture. A summary of the architecture and main features of GPUs is provided in Section 2.3, followed by an overview of field-programmable gate arrays (FPGAs) in Section 2.4. The ePython programming language is introduced in Section 2.5. Finally, Section 2.6 discusses related work for accelerating dynamic programming languages.

2.2 Micro-core processors

As discussed in Chapter 1, although new generations of CPUs have provided increased runtime performance, it has been at the cost of increasingly unsustainable power consumption. As shown in Figure 2.1, traditional CPU Instruction Set Architectures

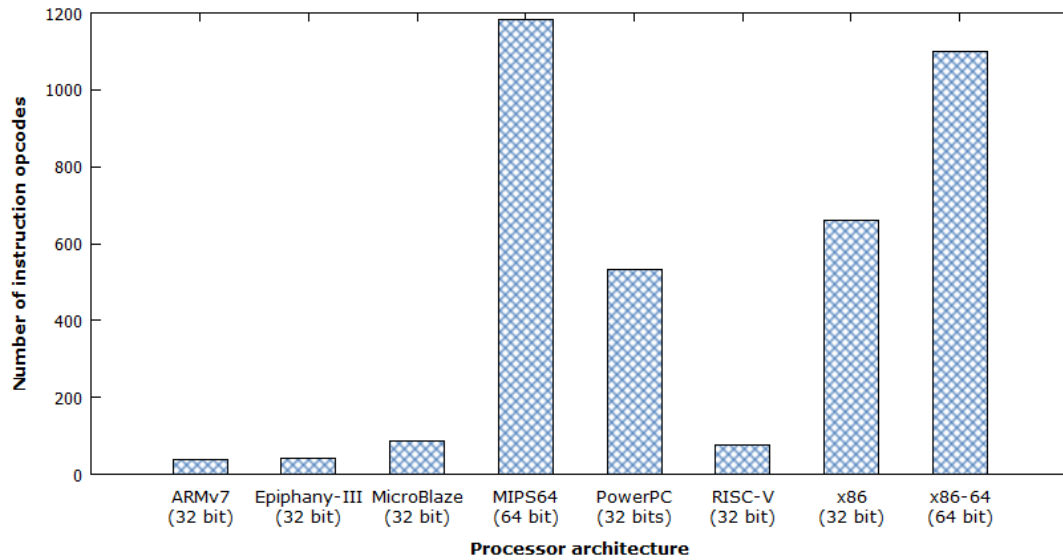


Figure 2.1: Processor instruction set size (number of opcodes)

(ISAs), such as those for the Intel x86, MIPS and PowerPC, are large in comparison to newer ISAs for the ARMv7, Epiphany-III, MicroBlaze and RISC-V. When we consider that complex ISAs with large numbers of instructions require more silicon area than simple ISAs [33] with a lower number of instructions, it would suggest that a large number of simple CPU cores would provide a better foundation for increasing runtime performance at low levels of power consumption. A possible drawback of simple cores is that they may not provide enough *richness* in their instruction set to satisfy current

application codes. However, as shown in Figure 2.2, the average ISA utilisation rate for the MIPS, PowerPC, Intel x86 and x86-64¹ ISAs for ten C benchmark applications that form part of the SPEC CPU2006 benchmark suite was between 5 and 20 percent for both the GNU Compiler Collection C compiler (*GCC*) and Portable C Compiler (*PCC*) [32]. When we calculate the number of instructions used by the GCC and PCC

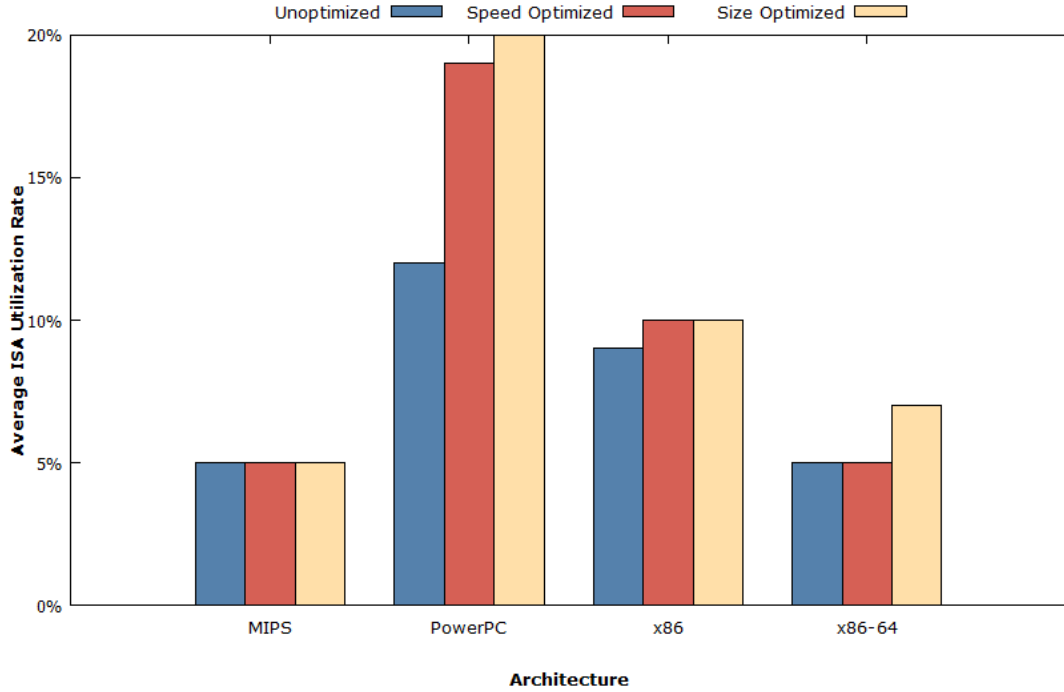


Figure 2.2: Average processor instruction utilisation by GCC and PCC compilers [32]

compilers, we find that between 59 (MIPS) and 107 (PowerPC) instructions are used for this SPEC CPU2006 benchmark subset. If we consider that the ARMv7, Epiphany-III, MicroBlaze and RISC-V ISAs shown in Figure 2.1 have between 40 (ARMv7) and 76 (RISC-V) instructions, and that the SPEC CPU2006 benchmark suite is supported on the ARMv7 [34], we can conclude that the large CPU ISAs are not mandatory, even for the speed optimisation (*-O3*) levels of the GCC and PCC compilers.

When comparing the ARM, Epiphany-III, MicroBlaze and RISC-V ISA size against the other ISAs, it should be noted that the x86, x86-64, PowerPC and MIPS64 ISAs include single instruction, multiple data (SIMD) / vector processing instructions, such as Streaming SIMD Extensions (SSE) and MMX [35] for the x86, AltiVec² [36] for the PowerPC and the MIPS SIMD Architecture (MSA) [37]. These SIMD instruction

¹Also referred to as x64, which will be used in the remaining chapters of this thesis.

²Also known as Vector Multimedia Extension (VMX) for IBM processors and Velocity Engine for Apple processors.

set extensions not only increase runtime performance but also significantly increase the instruction count. For example, a Fortran Finite-Difference Time-Domain (FDTD) method code was compared with a version that uses SSE *intrinsic*³ functions with the latter providing twice the runtime performance of the standard Fortran implementation [38]. Furthermore, [39] found that for a neural network code, SSE provided 3.7 times and MMX 4.1 times the runtime performance of the standard C code. However, Intel's SSE2, SSE3 and SSE4 extensions add a total of 204 instructions [40] to the x86 ISA, with MMX adding an additional 57 instructions [35], a significant increase considering the x86 instruction set is 659 instructions [32]. Moreover, these SIMD extensions often require the use of intrinsic functions [38] or assembly language code [39] to achieve the best runtime performance. For example, for a number of kernels, including the Sum of Absolute Differences (SAD), the Sum of Absolute Transformed Differences (SATD) and Discrete Cosine Transformation (DCT), the auto-vectorisation code of the GCC, *Clang* and Intel C (ICC) compilers only delivers, at best, 28% of the hand-tuned variants [41]. Considering that these three most popular compilers for the x86 / x86-64 ISAs are unable to generate code that leverages the full runtime performance the SIMD instructions, such ISA extensions could be considered as *instruction set bloat* as they consume valuable silicon area and increase both design complexity and power consumption [42].

The power consumption benefits of simpler cores, coupled with the fact that traditional CPU ISAs have instruction set bloat as demonstrated by [32], helps underpin the drive to create micro-core designs with a large number of simple cores on a single chip. The 16-core Adapteva Epiphany-III, shown in Figure 2.3, is an example of a micro-core design that provides 32 GFLOPS at a maximum power draw of 2 Watts, thereby delivering a GPU competitive 16 GFLOPs/Watt [43]. The 64-core Adapteva Epiphany-IV delivers a class-leading 64 GFLOPS/Watt [44] [9]. Micro-core architectures are not only of great interest for embedded, edge, and Internet of Things (IoT) applications, but also demonstrate potential in HPC, as discussed in Chapter 1. However, whilst competing technologies, such as GPUs, can deliver comparable GFLOPS/Watt, it is the low absolute power consumption of micro-core architectures that makes them appropriate for edge applications in the field, performing tasks such as Digital Signal Processing (DSP) [45]. For example, the 2W overall power consumption of the Epiphany-III is significantly lower than the average 194W for GPUs [46]. For edge applications that are often battery powered in the field, power consumption is critical

³GCC built-in functions to generate SSE code.

but there is often the runtime performance challenge where these applications have to execute within clearly defined time limits and micro-core architectures can deliver this balance of runtime performance and power efficiency. For example, although the Epiphany-III is relatively old (designed in 2013), it still delivers competitive runtime performance for DSP applications relative to specialised commercial hardware [12].

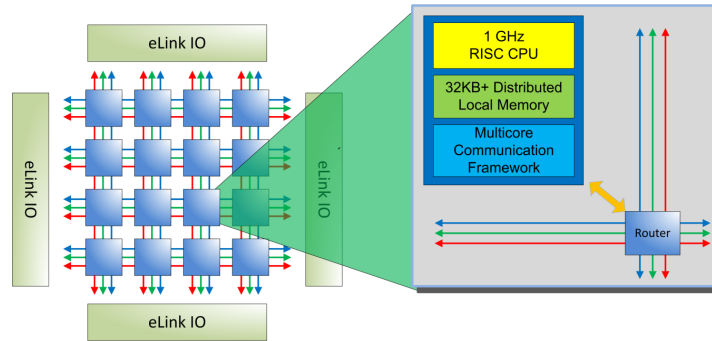


Figure 2.3: Adapteva Epiphany-III architecture [13]

Although micro-core accelerator architectures are attractive, they typically require device-specific knowledge and code, with kernel codes written in low-level C software development kits (SDKs). This increases the effort and understanding required to deploy applications to these architectures, especially when one considers the extremely small amounts of on-chip scratchpad memory (SPM) available, typically less than 64KB. Ideally, programmers should be able target micro-cores using high-level programming languages that abstract over the idiosyncrasies of the underlying architecture and provide significantly increased productivity over coding in low-level C, allied with sufficient runtime performance to enable the deployment of real-world applications on these technologies. A number of standard parallel programming technologies, for example OpenCL [47] and OpenMP [48], have been ported to micro-cores that help with the control and data transfer aspects of parallel programming. However, the OpenCL implementation is incomplete with partial support for the OpenCL 1.1 standard [49] and the OpenMP implementation implements a subset of the OpenMP 4.0 standard [50]⁴. Furthermore, they still require low-level C programming knowledge and an understanding of the bespoke architectural complexities of each micro-core target. A number of dynamic languages have been ported to micro-core architectures such as the Epiphany, including Erlang [52] and Lisp [53]. At varying levels of maturity,

⁴At the time of submission, the OpenMP implementation has been updated to be *adaptive*, whereby application-specific OpenMP runtime system (RTS) libraries deployed to reduce the memory footprint of the RTS and allow full OpenMP 4.0 support [51]

they have been implemented using interpreters, with the associated runtime performance overhead, but the problem remains that the critical memory limits of micro-cores severely impacts the viability of these approaches. Effectively, *we want to have our cake and eat it* [20]; the high-level parallel programming abstractions and productivity of dynamic programming languages, coupled with the runtime performance of technologies based on C (OpenCL, OpenMP and OpenACC [54]).

2.2.1 Micro-core architecture

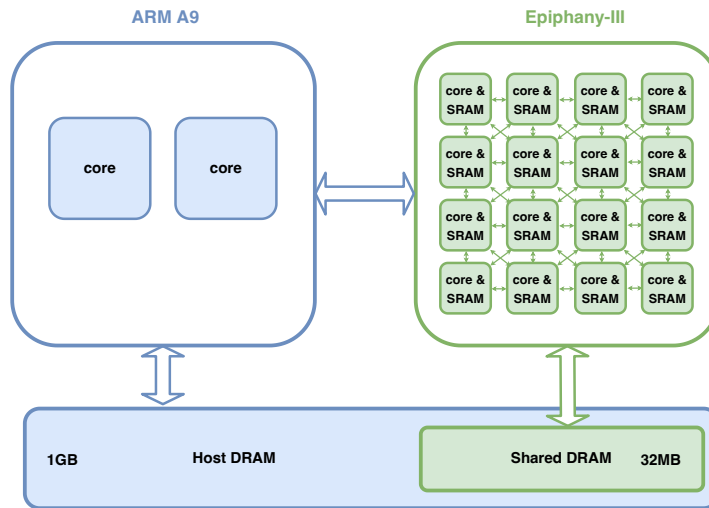


Figure 2.4: Typical micro-core architecture

Figure 2.4 shows a typical high-level micro-core architecture, in this example it is for the Adapteva Epiphany-III⁵. The Parallella single board computer (SBC) has a two-core ARM Cortex-A9, provided by the Xilinx Zynq-7000 series SoC (System on a Chip) and 16-core Epiphany-III micro-core processor. Each individual core has 32KB of SRAM (Static Random-Access Memory) but is able to access the SRAM of any other core on the Epiphany chip (for clarity, only neighbouring inter-core connections are shown) and the cores are able to access the off-chip shared 32MB DRAM (Dynamic Random-Access Memory) area within the host DRAM memory (1GB). The ARM cores are also able to access the memory of each Epiphany-III core via the *glue logic* within the Xilinx Zynq-7010 FPGA fabric, providing a form of UVA support⁶ on the

⁵The Cerberus FPGA micro-core design using the Xilinx MicroBlaze and RISC-V PicoRV32 soft-cores running on the Xilinx PYNQ described, in Section 5.2.1, is similar in architecture.

⁶The micro-core local memory is mapped into the host CPU's address space via a kernel device driver and the shared memory area on the host is mapped into the address space of the micro-cores.

Parallella.

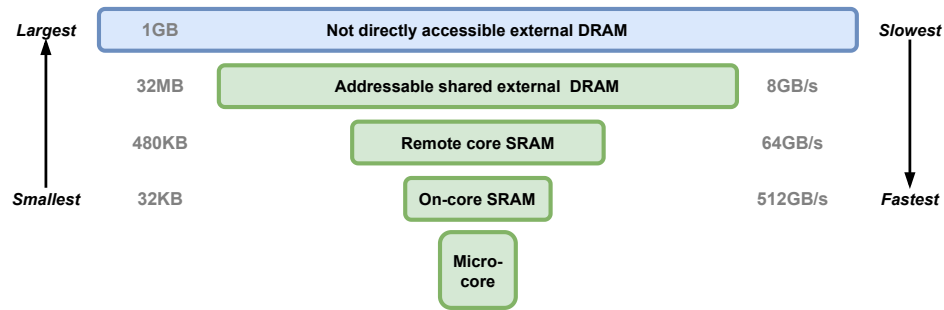


Figure 2.5: Adapteva Parallella micro-core memory hierarchy [22]

Figure 2.5 shows the *memory hierarchy* for the Adapteva Parallella, where the multiple levels of the hierarchy are increasingly larger but also increasingly slower as you move out from each micro-core. For example, the 32KB of local on-core SRAM is significantly faster (up to 64 times) than the 32MB of shared host DRAM. With no hardware caching support for the on-chip SRAM, significant programming effort is required to leverage the off-chip DRAM, whilst maintaining a high level of runtime performance for device kernels.

2.3 Graphics Processing Units (GPUs)

Whilst GPUs were originally designed to offload graphics operations for the display frame buffer from the CPU [57], they are now commonly used as General-Purpose GPU (GPGPU) parallel accelerators for scientific codes (including matrix multiplication and LU factorization [58]). GPUs, have a large number of *streaming multiprocessors* (SMs) containing multiple *scalar processor* cores. For example, as shown in Figure 2.6, the Nvidia Ampere GA100 has 128 SMs, each with sixty four 32-bit integer, sixty four 32-bit floating point and eight 64-bit floating point scalar processor cores. Within an individual SM, the binary is split up into *thread blocks* containing a number of *warps*, which are a collection of threads that execute the same instruction concurrently. This model whereby the same code (*single instruction*) is downloaded to the SM and different sections of the code are executed at the same time (*multiple threads*) is called Single Instruction Multiple Thread (SIMT) [59]. The model has limitations as each thread within a warp must execute exactly the same instruction per

However, the address space is not unified as the addresses referenced in the code on the host and micro-cores are different for the same physical address within the shared memory.



Figure 2.6: Nvidia Ampere GA100 GPU architecture [55]

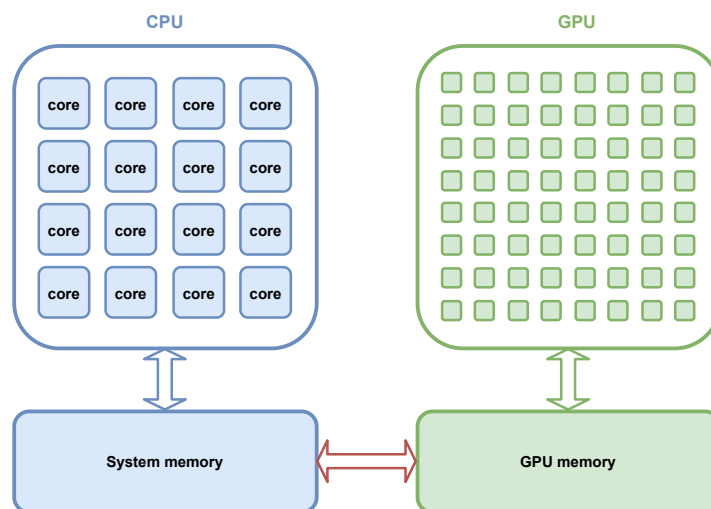


Figure 2.7: CPU / GPU memory architecture [56]

cycle (or that CUDA core is deactivated). Whilst this lockstep suits certain highly vectorised codes, the Multiple Program Multiple Data (MPMD) model of micro-cores is

more flexible, thereby supporting a larger number of general purpose codes.

Figure 2.7 shows the memory separation of typical CPU and GPU architectures. Historically, the programmer had to explicitly copy to and from the GPU memory using the underlying platform-specific APIs. Nvidia developed the CUDA [60] parallel programming platform to simplify the deployment of parallel codes to their GPUs, thereby providing a higher-level GPGPU (General-Purpose Computing on Graphics Processing Units) architecture. The management of the separate CPU and GPU memory address spaces was addressed with the introduction of supported Unified Virtual Addressing (UVA) in CUDA version 4. For example, if codes on the GPU contain pointers to an area of memory on the CPU, CUDA performs the transfers transparently with the associated non-local memory access performance overhead. The implementation of UVA is relatively basic and, as one might expect, repeatedly accessing CPU memory incurs a significant runtime performance penalty. CUDA version 6 introduced

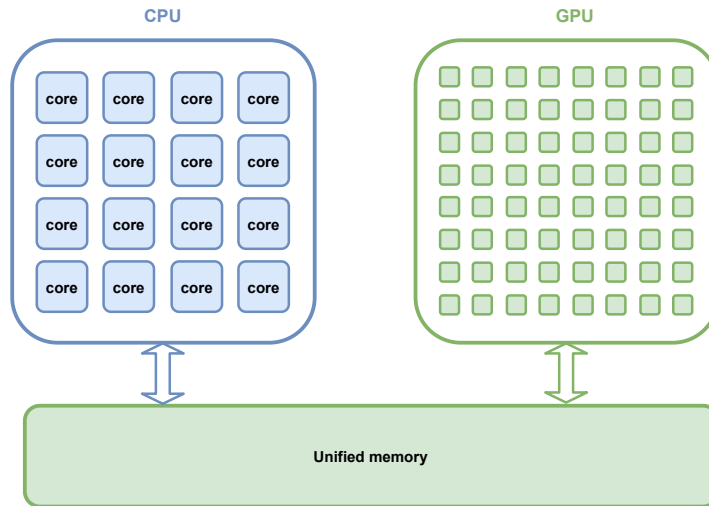


Figure 2.8: GPU Unified Memory Architecture (UMA) [56]

the Unified Memory Architecture (UMA), as shown in Figure 2.8, for Nvidia Pascal and later architectures. This builds upon UVA's virtual address space and provides additional support to transparently move memory to the GPU, effectively treating the GPU memory as a large cache. There are a number of memory movement strategies; the simplest strategy moves memory on a page fault, with more advanced strategies, such as pre-fetching, to minimise the non-local memory transfer overhead. Although the pre-fetching performance of UMA is comparable to explicit transfers into a pre-allocated buffer on the GPU using UVA, the latter is still the fastest mechanism to copy between the CPU and GPU [56]. As expected, these strategies are significantly

faster than using non-prefetched UMA transfers, due to the non-local access overhead (page migration) of the latter. Although this makes UMA attractive, the runtime per-

Table 2.1: Comparison of non-UMA and UMA programming models [61]

Step	Non-UMA	UMA
1.	CPU serial code and data initialization	CPU serial code and data initialization
2.	Data transfer to the GPU	Parallel kernel execution and implicit data transfer
3.	Parallel kernel execution	Synchronization between CPU and GPU
4.	Data transfer back to the CPU	CPU serial code
5.	CPU serial code	

formance benefits are dependent on the memory access patterns, with complex access patterns resulting in about a 2 times decrease in runtime performance [61]. Crucially, UMA also requires explicit hardware support within GPUs, with the resultant silicon area overhead and related power consumption increase [62]. Furthermore, the current implementation of UMA provides little additional developer productivity through the marginal improvement in code complexity when compared to the non-UMA programming model [61], as shown in Table 2.1.

2.4 Field-programmable gate arrays (FPGAs)

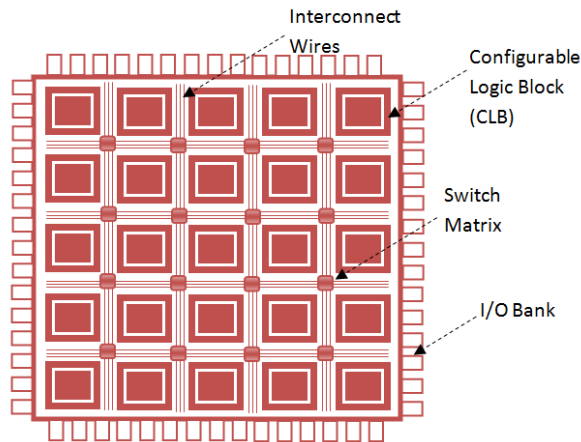


Figure 2.9: FPGA architecture [63]

FPGAs are user-programmable semiconductor devices that contain a matrix of logic blocks, as shown in Figure 2.9, that can be configured to create complex logic circuits, often replacing application-specific integrated circuits (ASICs), but are increasingly used as HPC accelerators. FPGAs are configured using a *bitstream*, generated by

the VHSIC Hardware Description Language (VHDL) or Verilog hardware description language (HDL). Whilst accelerator codes could be written directly in VHDL or Verilog, higher-level programming languages such as Xilinx High-level Synthesis (HLS) C [64] or OpenCL C [65] and toolkits such as SYCL [66] are more commonly used. The increasing use of FPGAs for HPC accelerators is driven by the fact that FPGAs use significantly less energy than GPUs for comparable levels of runtime performance. For example, [67] found that an OpenCL K-nearest neighbour (KNN) algorithm running on a FPGA used 10% of the energy of the equivalent code running on a GPU.

FPGAs can also be configured as processor cores, known as *soft-cores*, which can then be programmed like traditional CPUs. Furthermore, these soft-core designs can include application-specific extensions, such as the RISC-V Posit Enhanced Rocket Chip (PERC) that replaces the IEEE-754 2008 floating point unit (FPU) with a Posit Processing Unit (PPU) for better accuracy and precision [68]. Furthermore, larger FPGAs can support designs with multiple, interconnected soft-cores, using the available on-chip block RAM (BRAM) for local memory, thereby enabling the creation of micro-core processors similar to the Adapteva Epiphany-III discussed in Section 2.2. The proprietary 1680-core RISC-V GRVI Phalanx massively parallel accelerator is an example of such a design targeting HPC. The GRVI Phalanx has a stated peak throughput of 420,000 MIPS at a clock speed of 250MHz on a XCVU9P FPGA [69]. On a much smaller scale, the Cerberus soft micro-core processor presented in Section 5.2.1 is another example, created to support the research of this thesis. However, it should be noted that soft-cores have restricted clock rates, for example 250MHz for the GRVI Phalanx and 100MHz for Cerberus, limiting the runtime performance of each individual core in comparison to traditional *hard* CPU cores.

2.5 ePython

Dynamic programming languages such as Python [70], R [71], Ruby [72], JavaScript [73] and Lua [74] provide high-level, interactive programming environments. These languages have seen a significant growth in popularity over static, compiled programming languages such as C/C++, Fortran and Java [75] due to their greatly increased productivity, in terms of a reduced number of lines of code and time taken for the programmer to develop an equivalent solution [76]. The increased programmer productivity benefits led to the creation of ePython [19] that implemented a subset of Python 2.7 and, at the start of this research, specifically targeted the Epiphany-III. However,

designed with portability across micro-core architectures in mind, it has been ported to additional micro-core architectures and has evolved from its initial purpose as an educational language for parallel programming to its use as a research vehicle for understanding how to program micro-core architectures.

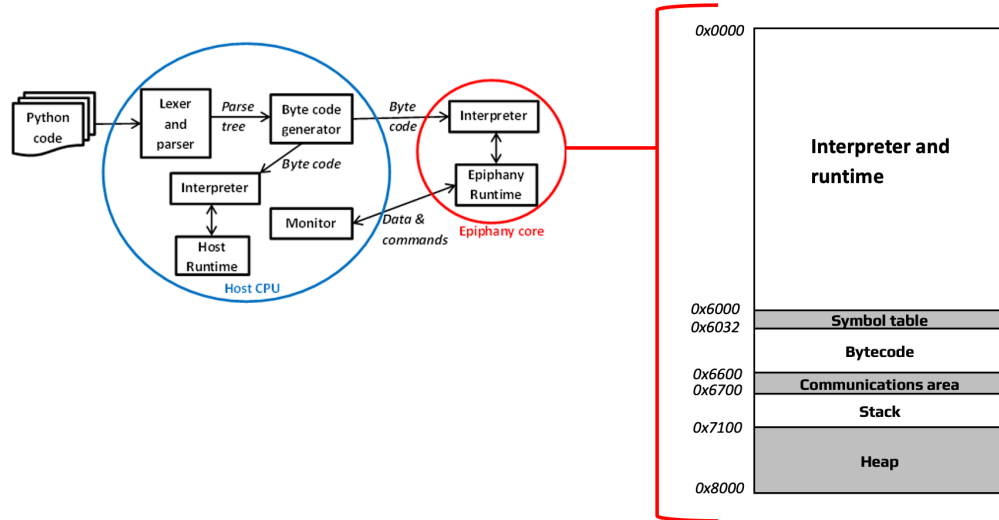


Figure 2.10: ePython architecture, including VM memory map [19]

The ePython architecture is shown in Figure 2.10. The components outlined in blue on the left of the diagram run on the host, including the Python compiler that generates bytecode and the monitor that manages kernel launch and communications. The ePython VM that runs on the device, consisting of the bytecode interpreter and runtime libraries, is outlined in red. As the VM also runs on the host, it is possible to run virtual cores to simulate larger devices. To the right of the diagram is the Epiphany-III memory map for each core, including the symbol table, communications area, interpreter stack and heap. Although the interpreter is tiny at around 24KB, it amounts to 75% of the total core SRAM on the Epiphany-III and after the symbol table, bytecode and communications areas are included, there is just over 6KB available for the stack and heap. This very small memory space available to kernel data severely restricts the applications that can be deployed to the Epiphany-III with ePython. However, it is possible for the bytecode, stack and heap to be placed in the 32MB of shared memory, with the associated runtime performance impact [22].

```
1 from parallel import reduce
2 from random import randint
3
4 a = reduce(randint(0,100), "max")
5 print "The highest random number is " + str(a)
```

Listing 2.1: Collective message reduction example [20]

Codes written in ePython can either be run *standalone* as shown in Listing 2.1, using the default virtual core running on the host for messaging between the host and device cores, or *offloaded* as kernels from within a Python script running on the host, as shown in Listing 2.2. The default ePython parallel programming model is single program, multiple data (SPMD), where a single kernel is deployed to all cores that each operate on different data. However, ePython also supports multiple programs, multiple datastreams (MPMD), where each core can execute a different kernel that processes different data. For MPMD, in the standalone mode, the placement of kernels on specific cores is controlled by the ePython `-c` command line argument, which can take a specific core or a range of cores for targeting the kernel. For offloaded kernels, the *function decorator*⁷, that marks the function as a kernel for offloading, provides the `target` parameter that allows the target core(s) for the kernel to be specified, as shown in Listing 2.2.

⁷Similar to *directives* in OpenMP [48] and OpenACC [54]

```

1 @offload(target=[4, 5, 6, 7])
2 def module1(LOOPS, BATCHES):
3     D1 = 0.3999999946405E-1
4     D2 = 0.96E-3
5     D3 = 0.1233153E-5
6     E2 = 0.48E-3
7     E3 = 0.411051E-6
8     s = 0.0
9     u = 0.0
10    v = 0.0
11    w = 1.0
12    x = 1.0/LOOPS
13
14    for count in range(BATCHES):
15        for i in range(LOOPS-1):
16            v = v + w
17            u = v * x
18            s = s + (D1+u*(D2+u*D3)) / (w+u*(D1+u*(E2+u*E3)))
19    return s

```

Listing 2.2: Example ePython offloaded kernel

The example in Listing 2.2 also shows how values can be passed from the host to the kernels running on the cores via function parameters (`LOOPS` and `BATCHES` in `module1()`) that perform an implicit data transfer. The offloading and data transfer model is described further in Section 3.2.1, which also details the additions to ePython created by this research to overcome the on-chip memory constraints. As part of the parallel programming support, ePython provides core-to-core and host-to-core communications via the `send()` and `recv()` functions, as well as *collective communications* via the `reduce()` function call with the operations `max`, `min`, `sum` and `prod`. The `bcast()` function is provided to broadcast a value from one Epiphany core to all the other cores [19]. With these functions, ePython programmers are able to create parallel codes that leverage the multiple cores of the Epiphany micro-core processor. Listing 2.1 illustrates a simple example of the `reduce()` function call with the `max` operation to determine the maximum random number between 0 and 100, displayed by each core.

The ePython VM and host-based monitor are written in C with portability across architectures a key design aim. As micro-core architectures have very limited memory,

the device code does not use the standard C library. Instead, the support and device-specific code is contained within the ePython runtime library. The result is that the C99 standard interpreter is portable across architectures, with only a new runtime library required for each new architecture, including the API calls for memory management and garbage collection, and communication between micro-cores and the host.

2.6 Accelerating Python codes

There are a number of technologies for offloading computation to micro-cores, including OpenCL [77], BSP [78], OpenMP [79] and MPI [80] on the Adapteva Parallella. However, not all are fully implemented, for example OpenMP and MPI. Furthermore, whilst these C-based technologies provide insight into the mechanics of kernel offloading and communications, ePython already provides these capabilities. Therefore, the focus of the investigations presented here are techniques to increase the performance of dynamic programming languages.

The Python programming language is, at the time of writing this thesis, the most popular programming language [75]. Whilst Python is very popular with programmers, its use of an interpreter results in performance significantly slower than statically compiled languages, such as C [81] and Fortran [82]. This has driven the need to overcome the performance overhead of the interpreter and the restrictions imposed by the global interpreter lock (GIL), which effectively makes all Python codes single-threaded by preventing multiple threads from executing Python bytecode at the same time [83]. This has resulted in technologies to increase the performance of existing, single-threaded Python codes through the compilation to native code, including Cython [84] and MicroPython [85]. The GIL limitation on multi-threading, coupled with the high-throughput parallel processing of GPUs, has resulted in work to leverage GPUs from within Python, including Numba [86], Copperhead [87], Parakeet [88], ALPyNA [89] and PyCUDA [90]. However, unlike Numba, Copperhead, Parakeet and ALPyNA, PyCUDA does not abstract the generation of GPU code but instead embeds CUDA C code directly within the Python source code. Therefore, it is more related to C-based technologies, rather than the approaches taken by the other technologies to accelerate Python-like codes on GPUs (and CPUs).

The high-level approach of Numba, Copperhead and Parakeet is similar, whereby they define an embedded domain specific language (eDSL) and utilise Python *function decorators* (directives) to annotate the code to be compiled to native code or offloaded

to GPUs. For example, Copperhead uses the `@cu` decorator to mark Copperhead language functions, Parakeet uses the `@PAR` decorator and Numba defines the `@jit` decorator to target CPUs, and the `@cuda.jit` decorator to target GPUs and perform the necessary data transfer. Listing 2.3 is an example of the Numba `@jit` decorator to generate native code for the `go_fast` function. The timing loop between lines 15 and 18, allied with the “DO NOT REPORT THIS...” comment on line 14 highlight that the JIT compilation time impacts the overall code runtime. The `nopython=True` parameter of the `@jit` decorator, hints to the Numba compiler that the function code can be executed outside the standard Python interpreter, thereby increasing runtime performance. Whilst Numba’s JIT compiler can speed up codes by a factor of around 20 times that of the standard Python interpreter [91], the generated code is larger than the memory available on micro-core architectures, necessitating a different approach for resource-constrained devices.

Although Copperhead uses a similar decorator approach to Numba, the eDSL uses function primitives to provide data-parallelism for single-dimensional arrays (vectors) of data. The primitives include `map()`, `reduce()`, `gather()`, `scatter()`, `replicate()`, `zip()`, `permute()` and `scan()` [87]. Parakeet also employs function primitives to provide data-parallelism on arrays, with codes in both languages appearing very similar, as shown by Listings 2.4 and 2.5.

```

1 from numba import jit
2 import numpy as np
3 import time
4
5 x = np.arange(100).reshape(10, 10)
6
7 @jit(nopython=True)
8 def go_fast(a): # Function is compiled and runs in machine code
9     trace = 0.0
10    for i in range(a.shape[0]):
11        trace += np.tanh(a[i, i])
12    return a + trace
13
14 # DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE
15 # EXECUTION TIME!
16 start = time.time()
17 go_fast(x)
18 end = time.time()
19 print("Elapsed (with compilation) = %s" % (end - start))
20
21 # NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
22 start = time.time()
23 go_fast(x)
24 end = time.time()
25 print("Elapsed (after compilation) = %s" % (end - start))

```

Listing 2.3: Numba JIT example [92]

Copperhead uses CodePy [93] to generate OpenMP C code for CPUs and PyCUDA to generate CUDA code for GPUs, achieving 45 to 100% of the performance of hand-coded CUDA code [87]. Parakeet, like Numba, uses LLVM [94] to generate code for CPUs and generates CUDA parallel thread execution (PTX) [95] code for GPUs, returning performance 2.4 times slower to 76% of that of hand-coded CUDA, where the majority of the overhead is due to PTX compilation [88]. Both technologies allow Python programmers to leverage GPUs to accelerate their codes without needing to understand the complex programming mechanics of CUDA and embed the accelerated code within their standard Python codes, providing greater accessibility and productivity than using alternatives such as PyCUDA.

```

1 @cu
2 def spmv_csr(vals, cols, x):
3     def spvv(Ai, j):
4         z = gather(x, j)
5         return sum(map(lambda Aij, xj: Aij*xj, Ai, z))
6
7     return map(spvv, vals, cols)

```

Listing 2.4: Copperhead example [87]

```

1 def add(x, y):
2     return x + y
3
4 def sum(x):
5     return parakeet.reduce(add, x)
6
7 @PAR
8 def norm(x):
9     return math.sqrt(sum(x*x))

```

Listing 2.5: Parakeet example [88]

ALPyNA adopts a different technique to generating GPU code than the eDSL and function decorator approach of Numba, Copperhead, Parakeet. Rather than requiring the programmer to select and annotate the Python functions that will be generated as GPU kernels, ALPyNA analyses loop data dependencies and performs automatic loop parallelisation to generate CUDA kernels for GPUs. ALPyNA leverages the constrained execution flow of Python loops, provided by iterator guarantee of the `range()` function and lack of a jump (*goto*) construct, to enable the static and dynamic analysis for auto-parallelisation [89]. ALPyNA only targets GPUs, using the LLVM-based Numba to generate the code. As Listing 2.6 shows, ALPyNA supports nested loop parallelisation and does not require hints or annotations, making it trivial for the Python programmer to take advantage of the performance increase of up to 1000 times that of the standard Python interpreter (CPython) [89].

```

1 def matmul(mat_a, mat_b, mat_c):
2     ma_rmax, ma_cmax = np.shape(mat_a)
3     b_rmax, mb_cmax = np.shape(mat_b)
4     for k in range(ma_cmax):
5         for i in range(ma_rmax):
6             for j in range(mb_cmax):
7                 mat_c[i][j] = mat_c[i][j] + mat_a[i][k] * mat_b[k][j]

```

Listing 2.6: ALPyNA example [89]

Whilst Numba, Copperhead, Parakeet and ALPyNA provide compelling examples of code generation technologies for CPUs and GPUs, they integrate with the CPython interpreter, restricting their use to traditional desktop⁸ and HPC systems, rather than extremely memory-constrained *bare metal* embedded, IoT and micro-core devices. MicroPython [96], a version of Python for embedded devices, implements native code generation on embedded CPUs via two *emitters*, *native* and *viper* [97], defining the `@micropython.native` and `@micropython.viper` decorators to select the emitter for native code generation. The *native* emitter lowers the MicroPython bytecode to machine code and performs calls to VM functions for operations such as arithmetic calculations, binary operations and comparisons. Effectively, this method removes the overhead of the VM's dispatch loop, whilst leveraging the existing VM functions and capabilities. The *viper* emitter takes this approach further by also generating machine code instructions for operations, rather than calling the VM functions. As expected, this increases performance yet further as arithmetic operations are performed inline rather than via a procedure call. This results in performance approximately 10 times that of the *native* emitter and around 24 times faster than the VM interpreter [98]. The MicroPython cross-compiler can also use the emitters to generate native code on the host, instead of on the device by the standard compiler. It should be noted that the emitters on the device are not JIT compilers as the native code is generated before execution begins and the bytecode is not profiled to select compilation candidates. For MicroPython's embedded targets, the latter approach allows a single MicroPython code file to be deployed to devices with different CPUs (e.g. ARM and Xtensa) where it is then compiled to the specific machine code of the CPU before execution, providing functionality similar to that provided by Java bytecode and the associated JIT compiler

⁸Here, the definition of desktop is any computer, irrespective of the processor or RAM, running an end-user operating system, such as Linux, Windows or macOS.

within the VM. Whilst MicroPython is specifically designed to target embedded devices, this approach precludes targeting micro-core architectures as it requires 256KB code memory for the VM and compiler, as well as a minimum of 8KB of RAM for the stack and heap [99].

The advent of multicore processors has shifted the burden of improving program execution speed from chip manufacturers to software developers.

Kulkarni et al., “Optimistic Parallelism Requires
Abstractions” [100]

3

Design of enhancements to ePython

3.1 Overview

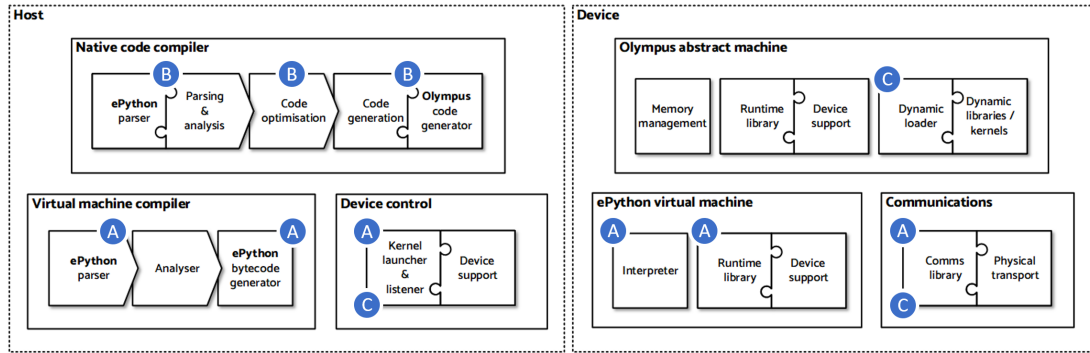


Figure 3.1: ePython architecture highlighting components discussed in this chapter

This chapter sets out the design decisions for the enhancements to ePython to address the key research questions: RQ1 (How to manage the limited on-chip memory for data?), RQ2 (How to manage the limited on-chip memory for code?) and RQ3 (How to address the low runtime performance of virtual machines?). Whilst some of the techniques are applicable to delivering dynamic languages on embedded devices, the focus here is on the design details required to provide programmers with capabilities to enhance the development of kernels in dynamic languages that are offloaded to the micro-core accelerator architectures described in Chapter 2.

Although there are a number of technologies for accelerating kernels written in Python on GPUs, such as Numba discussed in Chapter 2, ePython [19] is currently the only technology that supports offloading kernels written in a dynamic language to micro-core accelerators. While the techniques described in this chapter are applicable to dynamic languages more widely, the discussion will focus on extending and enhancing ePython as this is the primary research vehicle for this work.

Figure 3.1 shows the ePython architecture, including the existing virtual machine and new Olympus native code abstract machine components and compiler, highlighting those that are discussed in this chapter. Enhancements to the existing ePython memory model, **A** in Figure 3.1, are discussed in Section 3.2.2. Changes to the language to support native code generation **B** are examined in Section 3.4.1 and dynamic code loading **C** is covered in Section 3.3.1. The underlying implementation is discussed in Chapter 4.

3.2 Managing arbitrary data size (RQ1)

3.2.1 Domain Specific Language for micro-cores

Whilst ePython was originally developed as a language for teaching parallel programming on the Epiphany-III micro-core architecture [19], it was later developed into an eDSL within Python¹ running on the host to enable the offloading of kernels to micro-core accelerators [101]. Here, specific functions can be annotated as kernels for offloading to the micro-cores and there are a set of communication functions that allow the programmer to control the transfer of data to and from the host [102]. For example, Listing 3.1 illustrates how the `@offload` function decorator can be used to mark the `helloworld()` function for download to the micro-cores for execution. This example also shows the implicit transfer of data to and from the micro-cores; the values of the input parameters `a=10` and `b=20`, and the return value (a list of the value 30, one value for each core in the micro-core accelerator²).

```

1 from epython import offload
2
3 @offload
4 def helloworld(a,b):
5     print "Hello World"
6     return a+b
7
8 print helloworld(10, 20)
```

Listing 3.1: Synchronous ePython kernel launch example [102]

The `@offload` decorator can also be passed parameters to control the execution of the kernel. For example, in Listing 3.2, the `helloworld()` function is marked for asynchronous execution by the `async=True` argument. In this example, the kernel function returns an ePython *handle* that can then be checked to determine if execution has completed; the `handler.wait()` function call will block the host Python script's execution until the handle indicates that the kernel has completed. The `@offload` directive also has parameters that determines the set of cores (or a single core) that

¹The ePython eDSL can be embedded within any Python environment running on the host. As the standard Python interpreter written in C (CPython) was used for ePython, CPython and Python are synonymous in this work.

²In the case of a single Adapteva Epiphany-III processor, a list of 16 elements of the value 30 ($a+b \Rightarrow 10+20$) is returned i.e. `[30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30]`

will execute the kernel function, enabling different kernels to execute on the micro-cores at the same time. More information on the communication functions, including `copy_to_device()` and `copy_from_device()`, can be found in the ePython tutorials on GitHub [103].

```

1 from epython import offload
2
3 @offload(async=True)
4 def helloworld(a,b):
5     print "Hello World"
6     return a+b
7
8 handler=helloworld(10, 20)
9 print handler.wait()
```

Listing 3.2: Non-blocking asynchronous ePython kernel launch example [102]

3.2.2 Host / device memory model

The ePython eDSL provides the programmer with a rich set of features with which to write and control kernels deployed to the micro-core accelerators. However, the implicit data transfer of offloaded kernel functions imposes the restriction that the kernel cannot begin executing until the data transfer is complete. Whilst this might be expected, and is certainly acceptable for simple values, such as `a` and `b` in Listings 3.1 and 3.2, it becomes more problematic for large compound objects. These *copy on load*³ or *eager copy* semantics limit kernel runtime performance as they prevent the overlapping of communications and computation. Furthermore, whilst the eager copy model makes sense for GPUs, it necessitates that there is sufficient memory on the accelerator to hold the copied data [22]. When we take into account the tens of kilobytes of on-chip memory in micro-core devices, this reduces the viability of real-world applications on these architectures. For example, running the LINPACK benchmark [21] on the Adapteva Epiphany-III using the Eithne framework discussed in Section 5.2.2, the problem size had to drop by orders of magnitude from the default $N = 1000$ to $N = 50$ in order to fit within the micro-core’s on-chip memory [28].

³The required kernel data is copied over to the device before the kernel executes. Therefore, kernel execution does not begin until after the data is transferred.

3.2.2.1 Pass by reference

Not only does the eager copy model restrict runtime performance and the amount of data that can be passed to a kernel function, it also does not follow the standard Python *pass by reference* model for compound objects. Whilst CUDA UMA supports pass by reference for kernel functions [104], this is provided by explicit hardware support, as mentioned in Section 2.3. However, micro-cores do not have this hardware support and the default model is explicit copying. Therefore, this requires that the support is implemented in the programming technology to work around the lack of explicit hardware support. From a programmer's perspective, it would be preferable for eDSL kernel function calls to behave in exactly the same manner as standard Python functions running on the host. In other words, the kernel should be able to not only access passed compound object values but also be able to update them in place, per pass by reference semantics. Furthermore, the pass by reference model provides a mechanism whereby kernel functions can access host vectors that are much larger than the available on-chip memory; when a host vector element is referenced in the kernel, an implicit data transfer is performed to retrieve each element. Likewise, when an element is updated locally, the new value is sent back to the host to update the original vector. Without additional programmatic control, these data transfers block the device core execution until they complete [22]. Whilst this mechanism follows Python semantics and enables access to data structures larger than the on-core memory, the resulting runtime performance impact of transferring each element individually is undesirable. To address this issue, the *prefetching* of host vector *slices* was introduced by extending the `@offload` decorator's parameters. Figure 3.2 details the `prefetch` parameters, showing the relationship between the host vector, the data being prefetched and the local (on-core) vector slice.

The `prefetch` argument is keyed by the `parameter` name of the kernel function to which the prefetching configuration applies, `buffer size` determines the amount of memory allocated on the device for the local vector slice, `elements per prefetch` controls the length of each transfer slice, `distance` is the number of elements ahead of the current index that triggers the prefetch and `access modifier` (`readonly` or `readwrite`) determines if the local slice is *immutable*. The `access modifier` parameter is an additional optimisation that removes the requirement to automatically transfer the local vector slice values back to the host if they are marked *readonly* (immutable). Listing 3.3 shows the `prefetch` parameters for the example kernel function

```
@offload(prefetch={parameter name, buffer size, elements per prefetch,
                    distance, access modifier})
```

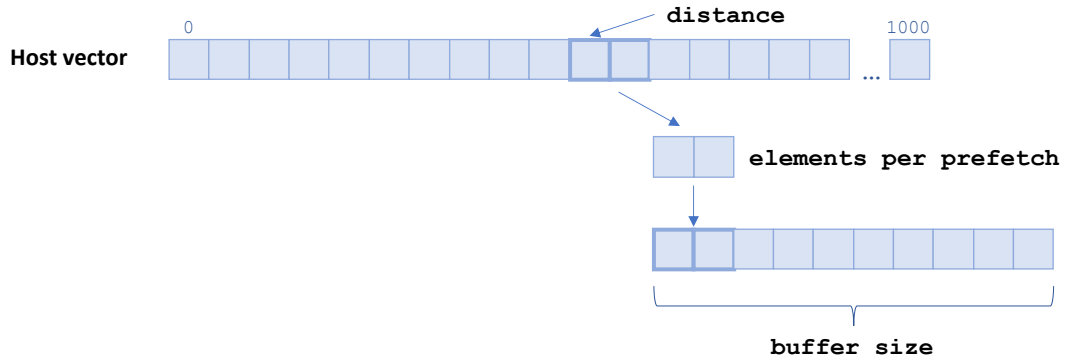


Figure 3.2: ePython `@offload` decorator prefetch parameters

`mykernel()`. Like the example in Figure 3.2, the `prefetch` parameters for `a` and `b` have a local size of 10 elements (buffer size), two elements will be fetched at a time for each vector (elements per prefetch), with elements being prefetched 10 elements ahead of the current indices (distance) and the local vector slices are immutable, negating the requirement to copy the local elements back to the host. When the kernel is first executed, the local slice will be preloaded with the first n elements of the host vector, where n is the buffer size.

Currently, the parameters are explicit to allow the programmer set the values and control the transfers. However, in future these parameters could be inferred by the underlying technology and set automatically to optimise the transfers, using similar SPM *tiling* techniques to [105] and [106]. However, the current ePython model demonstrates the general approach from a technology perspective, as well as the associated benefits of its implementation within a dynamic language running on micro-core architectures. Extending the model to include an automatic memory tiling approach is beyond the scope of this research.

The ability to prefetch host vector slices not only improves runtime performance by providing the opportunity to overlap communication and computation through direct memory access (DMA) transfers but also by transferring a larger block of data than for each individual vector index access, where significantly fewer messages are created that have to be responded to by the host⁴. For instance, on the Adapteva Epiphany-III,

⁴Depending on the messaging implementation, the host listener process may get swamped by a large number of individual messages, forcing the device cores to wait.

transferring larger blocks of data increases the overall effective bandwidth between the host and the device by up to 19 times for blocks of 8192 bytes versus blocks of 8 bytes [107]. Similar behaviour was observed by [80] for inter-core data transfers on the Adapteva Epiphany-III.

```

1 from epython import offload
2 import random
3
4 nums1=[0] * 1000
5 nums2=[0] * 1000
6
7 for i in range(1000):
8     nums1.append(random.randrange(0,100,1))
9     nums2.append(random.randrange(0,100,1))
10
11 @offload(prefetch={a,10,2,10,"readonly"}, {b,10,2,10,"readonly"})
12 def mykernel(a, b):
13     ret_data=[0] * len(a)
14     i=0
15     while i < len(a):
16         ret_data[i]=a[i] + b[i]
17         i+=1
18     return ret_data
19
20 print mykernel(nums1, nums2)

```

Listing 3.3: ePython prefetching example by annotating the `@offload` decorator [22]

3.2.2.2 Memory kinds

As discussed in Section 2.2.1, micro-core accelerators have a memory hierarchy, with increasing size and decreasing performance of the memory as you move out from the micro-cores (as shown for the Adapteva Parallella in Figure 2.5). The prefetching example in Listing 3.3 makes implicit use of this memory hierarchy, where objects in the non-directly accessible DRAM of the host Python interpreter’s heap are available locally to the kernel functions running on the micro-cores. However, it would be beneficial to provide fine-grained control over the allocation of objects within the memory hierarchy. For example, it would be useful to be able to access objects within the memory of the other cores on the device; on the Adapteva Epiphany-III, it is 27

times faster to load kernel codes onto a single core and from there distribute them to the other cores, in comparison to the standard method of loading the codes onto each core directly from the host [108]. To achieve required level of control over the area of the memory hierarchy within which objects reside, *memory kinds* [24] were introduced through the `memkind` ePython package that contains the `Host`, `Shared`, `Device` and `Local`⁵ classes that allocate objects in the non-accessible host memory, the shared off-chip memory, another core's memory on the device and the local on-core memory, respectively. Listing 3.4, although functionally identical to Listing 3.3, illustrates how the host level in the memory hierarchy can be explicitly selected by using the `Host` `memkind` class for objects `nums1` and `nums2`.

```

1 from epython import offload, memkind, types
2 import random
3
4 nums1=memkind.Host(types.int, 1000)
5 nums2=memkind.Host(types.int, 1000)
6
7 for i in range(1000):
8     nums1.append(random.randrange(0,100,1))
9     nums2.append(random.randrange(0,100,1))
10
11 @offload(prefetch={a,10,2,10,"readonly"}, {b,10,2,10,"readonly"})
12 def mykernel(a, b):
13     ...

```

Listing 3.4: Using memory kinds to control where objects are located [22]

The memory kind classes make it trivial for the programmer to change where in the memory hierarchy an object is allocated; all that is required is to change the `memkind` class to the one that manages objects in the desired area of the memory hierarchy. The underlying library and memory kind manages the low level details of this. Irrespective of where a variable is allocated, it is the reference that is passed to the micro-cores and the runtime manages the loads and stores. To create a kind representing a new level in the memory hierarchy requires a new Python class, inheriting from the `Kind` class, with all details about that level of hierarchy encapsulated inside the kind; everything else remains unchanged [22].

⁵Originally, the `Microcore` class allocated objects in the local SRAM of the micro-core [22] but it was replaced by the `Local` class when `Device` class provided access to other on-device memory.

3.3 Managing arbitrary code size (RQ2)

3.3.1 Dynamic code loading

Whilst ePython supports the execution of bytecode from within the off-chip shared memory area, there is a five times runtime performance penalty over running the bytecode from the on-chip SRAM [19] and the mechanism does not support dynamic function loading. In order to maximise the precious on-chip memory, it would be beneficial to be able to dynamically load and unload functions from within kernels. This is achieved by introducing a new Python decorator, `@dynamic`, which has the same functionality as the `@offload` decorator but indicates to the compiler that the function should be marked for dynamic loading. Listing 3.5 shows the use of the `@dynamic` function decorator, where the `add` and `add_nums` functions marked for dynamic loading. This example also demonstrates that functions can be dynamically loaded at declaration or any point later in a kernel's execution through the use of the `defer=True` decorator argument.

```
1 from epython import dynamic
2
3 @dynamic(defer=True)
4 def add(x,y):
5     return x+y
6
7 @dynamic
8 def add_nums():
9     global add
10    add = fetch_function("add")
11    print(add(3,4))
12    del(add)
13
14 add_nums()
```

Listing 3.5: Dynamic loading function declaration example [27]

The ability to delay the loading of dynamic functions provides the programmer with a high degree of control over the memory usage of functions within a kernel; dynamic functions are loaded into space allocated within the runtime's heap and can be marked for deletion by using the Python `del()` function, as shown in `add_nums()` in Listing 3.5. In contrast, existing ePython functions continue to consume memory after

they are loaded. Whilst there is no implicit support for dynamic function prefetching⁶ and deletion, the first-class nature of Python functions means that as long as a dynamic function is loaded and assigned to a reference before it is used, the loading can take place anywhere within the execution of the kernel that the programmer deems most beneficial. The ePython `fetch_function()` call is blocking and kernel execution waits until the function is fetched from the host, heap memory is allocated and the function loaded into the heap block. However, the alternative `request_function()` is non-blocking and returns a handle which can then be interrogated to determine if the function has been transferred and can be loaded into on-chip memory with `load_function()`⁷.

The `@dynamic` function decorator abstraction drives the compilation technology to support dynamic function loading within ePython kernels running on micro-cores. As Sections 6.4.1 and 6.4.2 demonstrate, this abstraction is an important first step that provides valuable capabilities for codes running on micro-cores and future work could automate this further, such as dynamic function prefetch and auto-deletion from memory.

3.4 Increasing kernel runtime performance (RQ3)

3.4.1 ePython object typing

Python is a dynamically typed language where objects⁸ can change their type throughout the program's execution. For example, an object instantiated with an integer literal can become a string or a list at a later point in the code. In contrast, statically typed languages, such as C, Fortran and Java, once an object is declared as a particular type, a compiler error will be raised if an attempt is made to change its type later in the code⁹. Like Numba [86], the ePython compiler performs *type inferencing* where object types, including function parameters, are determined from their surrounding environment. For example, it can be inferred that a variable is a floating point (real) number if it is

⁶Functions marked with the default `@dynamic` decorator that are not deferred are prefetched by the ePython runtime at kernel start-up.

⁷Originally, `load_function()` performed a blocking fetch from the host and load into on-chip memory [27].

⁸For the sake of simplicity, all source programming language elements, for example, numbers, strings, lists, arrays and functions will be referred to as *objects*.

⁹Accepting any implicit *type coercion* rules.

declared with a real number literal or a function's parameters are integers if integer literals are passed to a specific function call. However, the type inferencing is performed *statically* by the ePython compiler by traversing the Abstract Syntax Tree (AST) at compile time, rather than *dynamically* at execution time as performed by the Numba JIT compiler [109]. It is important to note that ePython objects retain their fully dynamic nature, in line with standard Python behaviour. This dynamic nature does not impose any undue complications for the implementation of an interpreted version of the language but it becomes problematic when the code is statically compiled to native code. One approach is to tag each object with its current type and dynamically interrogate this tag on every object access and perform the necessary memory operations based on the this type information. This is the method implemented within the ePython interpreter but it introduces an overhead that would be preferable to remove in order to maximise the benefits of native code generation. Ideally, in order to minimise the runtime performance gap between statically typed programming languages and natively compiled dynamic languages, dynamic type checking should be removed as much as possible. This requires that objects are statically typed checked, with the associated restriction imposed on codes that use dynamic typing. Whilst for most target scientific kernels this is not an undue restriction as they have *monomorphic*¹⁰ objects, it might appear to be a significant restriction for other Python codes. However, considering the *Awesome Python* list of applications [110] where approximately 80% of the objects in the code are monomorphic [111], it is reasonable to conclude that constraints on dynamic typing in Python can be made in order to improve runtime performance. Furthermore, although objects might be statically typed, dynamic type checks removed and typed object access introduced, type changes can be supported by creating new type declarations and updating the typed access when an object's type changes.

3.4.1.1 Type inferencing

The underlying approach used in this work, is similar to *gradual type inferencing* [112], *simple type freezing* [113], and the type inferencing performed by [114] and [115]. *Polymorphic* compound types, such as Python lists are more problematic as each element can be of a different type, thereby requiring type tagging of the elements, unlike a traditional vector or array where all elements are of the same type. The simplest solution for these data types would involve *boxing* [116] the elements along with their type

¹⁰An object is monomorphic if its type does not change during the execution of the code after it has been declared or assigned an initial value.

and performing type checking at runtime, with the associated runtime performance and storage overheads¹¹. As the primary focus of this work is accelerating scientific kernels, support is provided for typed vectors and arrays, with their associated runtime performance and storage benefits, rather than generic lists.

3.4.1.2 Dynamic typing and polymorphism

Whilst it is simple to infer the type for literals, function definitions are more problematic. For example, the compiler is unable to infer the types for the `x` and `y` parameters of `add()` function in Listing 3.5 from the declaration. However, it is possible to infer from the `add_nums()` function that they are both integers for this code. This information can then be used to type the `x` and `y` parameters as integers for code generation. If `x` had a value of 3.3 instead, the compiler could infer that `x` was a real number and type the `add()` function declaration accordingly. As Python is a dynamically typed language, it is possible for `x` and `y` to be any type that supports the `+` operator, for example, integers, reals and lists. This means that `add()` is an implicit parametric polymorphic function [117], adding complexity to the mechanism of statically typing functions for code generation as the compiler needs to consider all combinations of a function's input parameter types and the associated return types. The solution is to implement a form of *type scheme* for function *signatures* (input parameter and return types) described by [118], where a *template* is created for each function body or *lambda* (λ) and its signature, which is then copied for each function application or *beta* (β) *reduction*. During the type inference phase of compilation, for all β reductions, the argument types are substituted in the template and then the λ is typed to determine the return type. The typed signature and λ are then added as a *variant* to the function's definition. During code generation, these typed variants are used to generate the code that is executed for each combination of function argument types, with the generated function name *mangled*¹² to include the input parameter and return types, thereby supporting the polymorphic nature of dynamically typed functions. This process is transparent to the programmer, with the only overhead of increased binary size

¹¹Even if we assume a single byte for each element's type tag, this amounts to a 25% memory footprint overhead for integer values.

¹²The name mangling scheme is simple and leverages the fact that Python does not allow the `$` character in object names, unlike C99 Universal Character Name identifiers [119]. The generated name starts with `oly_`, then the return type, followed by a `$`, then the ePython function name, another `$` and the argument types. For example, the `add()` function that takes two integers and returns an integer would have the following mangled name: `oly_i$add$ii`.

for codes with a number of function variants.

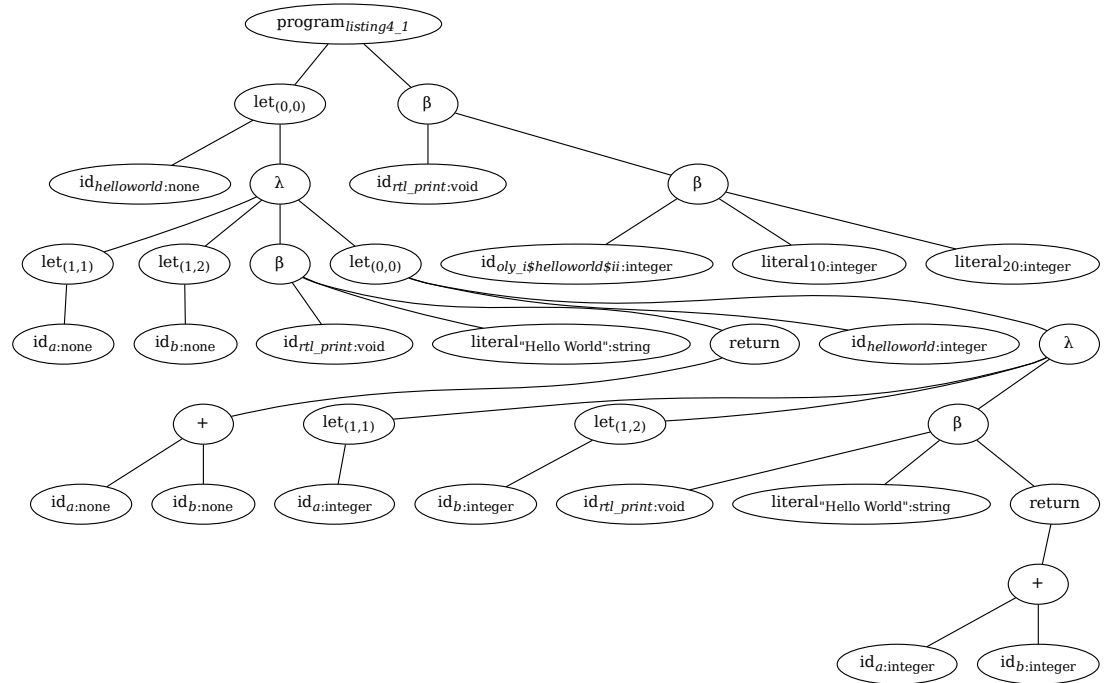


Figure 3.3: ePython AST example

Figure 3.3¹³ shows the AST generated for the example in Listing 3.1. In the Olympus AST, all λ s are stored on the left-hand side of the tree, with all other nodes chained on the right-hand side. The declaration node for the `helloworld()` function with the initial, untyped λ shown by the tree with `id` nodes `a` and `b` with type `none` can be seen on the left of the diagram. The typed variant is on the right and below, with the typed (`integer`) `a` and `b` parameters and `return` type. The typed β reduction is shown on the right-hand side of the AST where it is applied to the two integer arguments (`10` and `20`). The diagram also shows the mangled function name (`oly_i$helloworld$ii`) in the left-hand `id` of the β node, representing the variant of the `helloworld` function that takes two integer parameters and returning an integer value. As the creation of λ variants is driven from the β reductions, the original, untyped template λ is never used to generate code, and no λ code bodies are generated if no corresponding β reductions are performed.

¹³The AST diagram was generated using the Olympus `dotgen` code generator, which creates a Graphviz `.dot` file directly from an ePython source file. This can then be rendered by Graphviz in a number of image formats, including EPS, PDF, SVG and PNG.

```
1 def inc(x):  
2     if x < 100:  
3         return x+1  
4     else:  
5         return "Too big"  
6  
7 def dec(y):  
8     if y > 0:  
9         return y - 1
```

Listing 3.6: Multiple function return types

Without discussing the merits of the `inc` function Listing 3.6, it should be noted that the proposed type inferencing design does not attempt to address Python functions with polymorphic return types, even though there are possible use-cases for constructor functions that return different value types. As ePython does not currently support classes / objects and to reduce algorithmic complexity within the type inferencing phase, the Olympus ePython compiler does not implement *dynamic fallback* [112] for instances when it cannot infer object types and will return an error for cases such as that shown in Listing 3.6¹⁴. However, it should be noted that return types are *coerced*, such that an ePython function that returns both integer and real values will have a return type of real, as integers are *promoted* to reals by the type inferencer when coercing the two types. Furthermore, to avoid dynamic fallback, [112] propose a strategy for evolving dynamically typed programs to statically typed programs:

“ In one state, type inference removes unnecessary assumptions of the dynamic type, so that elimination of any remaining dynamic types in the code requires either reasoning outside the static type system, or restructuring the code. In the other state, the programmer either introduces further type annotations or restructures parts of the program to conform to the static type discipline. ”

This strategy can also be applied to ePython codes, whereby the second state can be used to allow the type inferencer to determine static types for program objects. For example, the return type of the `dec` function in Listing 3.6 is ambiguous as it can either be a numeric value when `dec` is greater than 0 or `None` otherwise. Whilst this

¹⁴However, the current approach does not preclude implementing dynamic fallback in the future, through the introduction of a `dynamic` type within the Olympus abstract machine and the requisite boxing of the return values.

is the same behaviour as Python, it is still problematic for later use of the returned `None` value. In this example, the addition of the line `return 0` at the end of the `dec` function allows the type inferencer to determine a static return type (integer). In the case of the Olympus abstract machine, if `y` is real, the final integer return value (0) will be promoted to a real and the corresponding real variant added to the λ node. For most codes, especially scientific kernels, the Olympus approach, allied to strategy of [112], provides behaviour close to that of Python, whilst greatly simplifying the type inferencing algorithm.

Supporting the required functionality [on micro-cores], which was originally designed for shared-memory multiprocessors, can be a very difficult procedure due to limited resources.

Agathos and Dimakopoulos, “Adaptive OpenMP Runtime System for Embedded Multicores” [50]

4

Implementation of enhancements to ePython

4.1 Overview

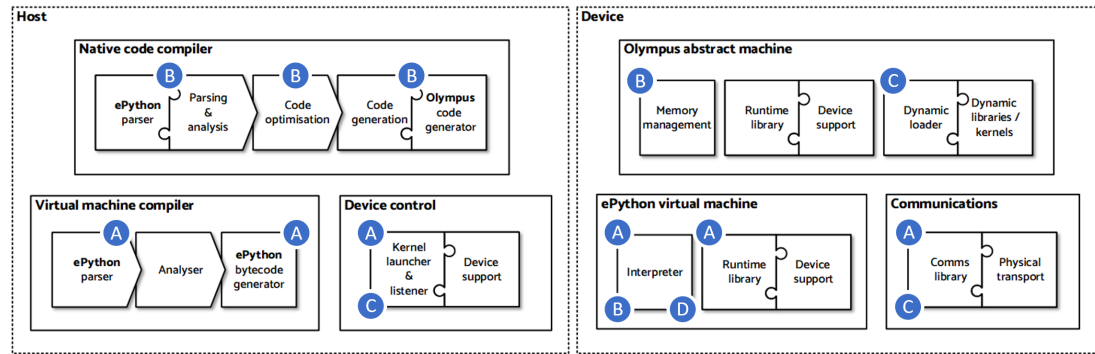


Figure 4.1: ePython architecture highlighting components discussed in this chapter

This chapter sets out the implementation of the enhancements to ePython to address RQ1 (How to manage the limited on-chip memory for data?), RQ2 (How to manage the limited on-chip memory for code?) and RQ3 (How to address the low runtime performance of virtual machines?).

Figure 4.1 shows the ePython architecture, including the existing virtual machine and new Olympus native code generation abstract machine components and compiler, highlighting the components that are discussed in this chapter. The support for arbitrary sized data (RQ1), **A** in Figure 4.1, is examined in Section 4.2, including the communications model and memory hierarchy implementation. The techniques to increase runtime performance (RQ3) through enhancements to the VM and the introduction of native code generation (Olympus), including compiler structure, runtime memory model and type management **B**, are discussed in Section 4.3. The dynamic function loading model developed to support arbitrary kernel code sizes (RQ2) **C** is described in Section 4.4. The investigations into the impact on runtime performance (RQ3) and bytecode size of different addressing modes within a VM, supported by the creation of the new Merlin virtual machine **D**, are discussed in Section 4.3.2.

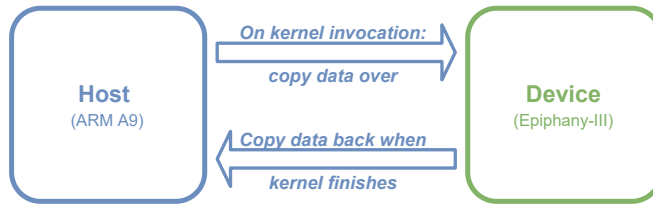
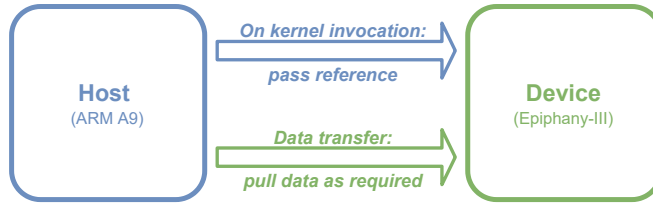
4.2 Managing arbitrary data size (RQ1)

This section details the enhancements to ePython to support the ability for kernels to escape the restrictions on data size that the limited on-chip memory of micro-cores impose. A brief comparison with GPUs, will be provided to help set the context and background for the ePython enhancements. The discussion will then focus on how the pass by reference kernel data model (Section 3.2.2.1) can be delivered on micro-core architectures, including: the limitations imposed by the CPython interpreter on the host, the new protocol required to support the pass by reference kernel data model and the underlying functions required to support the design. As the memory hierarchy and memkinds, discussed in Section 3.2.2.2, are vital in providing the programmer with a high-level set of features to manage memory constraints, the implementation changes to support them will also be discussed, even if these are relatively trivial once the pass by reference kernel data model is implemented.

4.2.1 Communication model

As mentioned in Section 2.2.1, implementing a form of unified virtual addressing (UVA) on micro-core architectures requires both hardware and software support. On both the Adapteva Parallella and Xilinx PYNQ boards the hardware support for this UVA-like memory model is provided via IP blocks within the FPGA fabric of the Zynq-7000 SoCs and software support within the micro-core runtime support executing on the host. The support for UVA in ePython is provided by a set of runtime communication *primitives*¹ (Section 4.2.4) using *mailboxes* [120] to support communications between the micro-cores and the host. As shown in Figure 4.2, the original kernel data transfer model was to explicitly copy the data across to the micro-cores at kernel execution time and back to the host on kernel exit, in the same manner as the non-UMA GPU programming model in Table 2.1. Whilst this *eager communication* model is simple and well understood, the disadvantage is that the kernel has to wait until the data has been transferred before execution can start. Ideally, we would like to initiate kernel execution as soon as possible and transfer the data as it is required in a manner similar to UVA “Zero-Copy” memory [104] or UMA. Here, a reference to the data is passed to the device at kernel invocation and the data is pulled from the host using this reference, as shown in Figure 4.3. The advantage of this *lazy*

¹A set of *native functions* that implement low-level functionality to provide a base for building higher-level abstractions for the programmer.

Figure 4.2: ePython *copy and launch* communications modelFigure 4.3: ePython *reference and pull* communications model

communication model is that it matches the Python pass by reference semantics for function calls, where function parameters that are compound objects, such as lists, are references (pointers) to the original objects, rather than copies made on entry to the function. Crucially, this model allows access to the host memory² that is significantly larger than the tiny amounts of micro-core on-chip memory. The underlying mechanism for passing references to host Python heap objects to device kernel functions will be discussed in Section 4.2.2 below.

4.2.2 Host Python integration

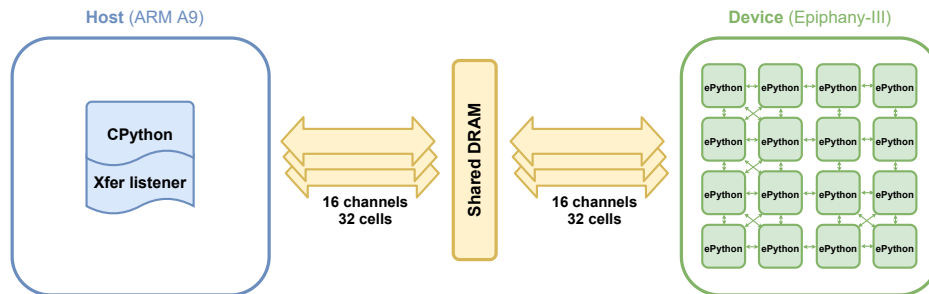


Figure 4.4: ePython communications architecture for the Adapteva Parallella [22]

In contrast to the static copy and launch model described in 4.2.1, where the data is simply copied to the shared DRAM on the function call, the pass by reference model

²This is not direct physical access to the host memory from the device kernels but is provided by the host / device communications discussed in Sections 4.2.2, 4.2.3 and 4.2.4.

is dynamic, requiring a new communications architecture, as shown in Figure 4.4. The implementation for ePython consists of a CPython module on the host containing the Python interpreter interface and host / device communication functions. The module also includes a *pthread* [121] listener that responds to requests from the device kernels. This listener thread only acquires the CPython interpreter's Global Interpreter Lock (GIL) [83] while it is accessing Python heap objects in order to minimise the impact on the host Python script and to increase the responsiveness of the listener to device requests.

An area of the host shared DRAM, shown in Figure 2.4 and Figure 4.4, is used for the communication *channels* between the host CPU and the micro-cores, typically this is one per core, although for large core counts these can be multiplexed. Each channel has thirty two³ 1KB *cells*, enabling up to thirty two concurrent transfers between the host and each of the micro-cores, with larger data payloads occupying multiple cells at the expense of the number of concurrent transfers. The host module allocates the communications channels and cells in shared memory using the `allocateSharedBuffer()` function, which is unique for each micro-core architecture. For example, the Adapteva Parallella version uses the `e_alloc()` function from the Epiphany Hardware Utility Library (eLib) [122] and the Xilinx PYNQ version, for the MicroBlaze and RISC-V, uses the `cma_alloc()` function from the Contiguous Memory Allocator (CMA) library [123], that allocates an area of shared memory visible to both the host processor and micro-core device. Both these libraries provide an abstraction over the specific, low-level implementation of the device driver within the Linux kernel.

4.2.2.1 CPython integration limitations

The Python module interacts with the CPython interpreter and, similar to ALPyNA [89] and Parakeet [88], provides access to a subset of Python types (scalars and lists) within the interpreter's heap⁴, with the limitation that scalar variables cannot be updated from within device kernels. This is in line with the standard Python programming model as scalars are *immutable* in Python, whereby updates to a scalar do not change its value but point it to a new object containing the new value. Within the CPython interpreter, all variables of the same value point to the same object in mem-

³This number is configurable based on the amount of shared memory available. It was arbitrarily set at thirty two for the Adapteva Parallella as 32KB is the maximum amount of SRAM per core.

⁴As this functionality is provided by a Python module written in C, it would be possible to also provide access objects external to the host Python interpreter's heap.

ory and when their value is updated the variables change ID (reference) to point to the new value object, as shown in Listing 4.1. In this example, the IDs for `x` and the value 2 are the same (9780832) and the ID for `y[2]` changes from 9780832 to 9780896 (the ID for the value 4) after the line `y[2] = 4`. However, as expected, the ID for the list `y` remains the same after element 2 is updated as lists are *mutable*. Therefore, whilst we can pass scalar variables to the device kernel functions, they cannot be updated even if we wished to deviate from the standard Python programming model. Since lists are mutable and the Python C API [124] allows us to create new scalar values and update the lists stored in the host Python interpreter’s heap, we can pass and update data using lists.

```

1 x = 2
2 print(id(x))           # prints 9780832
3 print(id(2))           # prints 9780832
4 x += 2
5 print(id(x))           # prints 9780896
6 print(id(4))           # prints 9780896
7 y = [ 0,1,2,3 ]
8 print(id(y))           # prints 139692247033344
9 print(id(y[2]))        # prints 9780832
10 y[2] = 4
11 print(id(y))           # prints 139692247033344
12 print(id(y[2]))        # prints 9780896

```

Listing 4.1: Python scalar immutability and IDs

4.2.3 Communication protocol

A low-level communications protocol, *Xfer*, was designed to support the pass by reference model discussed in Section 4.2.1. The protocol uses a *control header* that contains the *operation*, *variable ID*, *variable type*, *size / length*, *message originator* and *message target*, followed by the data. The Xfer protocol operations are listed in Table 4.1. The operations marked in *italics* are extensions to the Xfer protocol to support the Eithne benchmarking framework discussed in Section 5.2.2. The communication requests consist of matched pairs of messages i.e. a request is made by the originator and a new message is created by the target in response. For example, a GET request from a device micro-core for an external variable’s value is matched by a SET message from the host containing the data. As mentioned in Section 4.2.2, each message occupies

Table 4.1: Xfer communication protocol operations used to support ePython VM memory hierarchy access, pass by reference kernel data model and Olympus abstract machine host / kernel communications

Operation	Description
CLEARED	Channel cell is cleared and available for transfers
GET	Request the value of a variable
SET	Message contains the value of a variable
REGISTER	Register an external variable for transfers
READ	Reserve the cell for a subsequent GET operation
WRITE	Reserve the cell for a subsequent SET operation
DEBUG	Turn on debug output from Xfer monitor
TERMINATE	Request from host to terminate device kernel
LOAD	Dynamic function load request

one or more 1KB cells within a channel, with both the device and host writing to the same cell(s) for a single request message pair. Request and response messages cannot be written to an Xfer cell unless the operation field contains **CLEARED**, preventing the cell being reused before processing is complete. If a new communications request cannot be satisfied by the number of free cells, the request is blocked until the required number of cells are available. The message flow between the device and host for an external (host) variable data request is shown in Figure 4.5. This flow describes a sim-

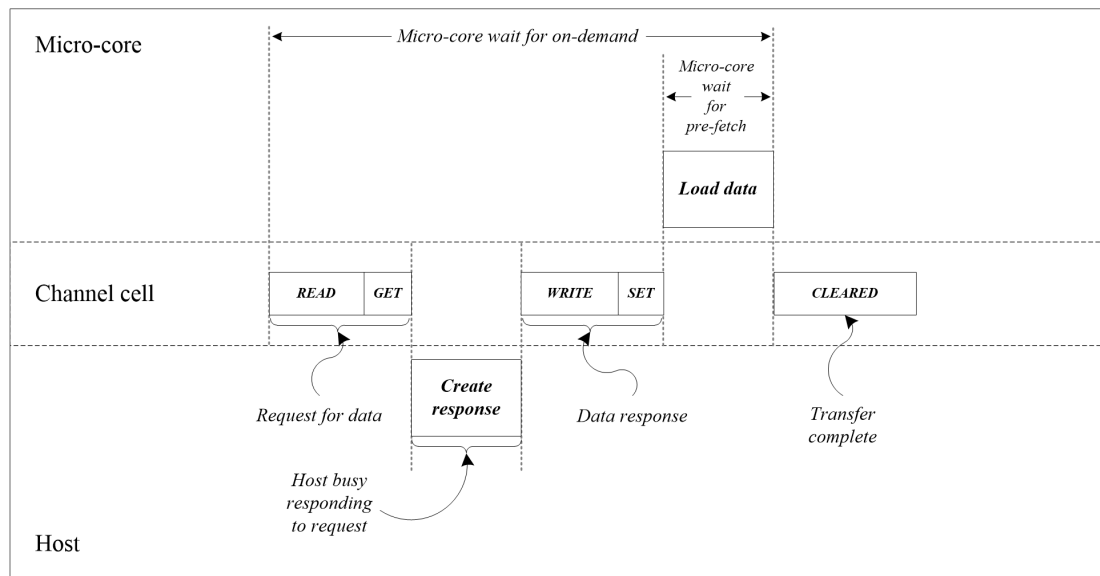


Figure 4.5: Xfer external variable data request message flow

ple *blocking* external variable request where the device waits until the data is returned

from the host before continuing execution. In this example, the micro-core is requesting the value of an external variable stored on the host. The micro-core writes a `READ` operation to the next available channel cell to reserve it. It then creates the request for the variable value and once written to the cell, it updates the operation to `GET`. In a blocking request as shown in Figure 4.5, the micro-core will now poll the cell operation (*blocks*). The host listener will respond to `GET` request and will update the cell operation to `WRITE`, reserving the cell, lookup the variable value and write it to the cell and then update the operation to `SET` when complete. The micro-core will now load the variable value into local memory and update the cell's operation to `CLEARED` when complete, releasing the cell for future transfers.

The protocol also includes the `REGISTER`, `TERMINATE` and `DEBUG` operations that register external variables on the host, terminate kernel execution and enable / disable debugging in the Xfer monitor program. The `LOAD` operation is used both by the Eithne framework and the Olympus native code abstract machine to support dynamic function loading (Section 4.4).

The CPython variables passed by reference to kernel functions are referred to as *external* variables in ePython as they are external to the ePython heap on the micro-cores. These variables are marked by the `share_with_device()` Python primitive that underlies the `@offload` function decorator. The mapping between the host CPython and device ePython variable IDs is held on the host to save the precious memory on the micro-core device⁵ and to simplify the communication calls. The mapping is created when the kernel functions are downloaded to the micro-cores and additional external variable registration calls are made that include the CPython variable IDs which are then returned by `REGISTER` messages, created by the device runtime, containing the corresponding ePython variable IDs. For the ePython interpreter, the symbol table was extended to include a flag to indicate if the variable reference is local and directly accessible, or if it is external (on the host) and non-directly accessible. This flag is then checked during kernel execution and if zero, the variable is accessed directly as before but if the flag is non-zero a `getExternalVariable()` call is made that sends an `GET` message to the host listener to retrieve the data, if required. For Olympus kernels, there are specific external variable access macros that lookup the variable ID and perform the `getExternalVariable()` call, if required. The listener looks up the ID mapping table, retrieves the data and returns an `SET` message containing the data to the device.

⁵In fact, the changes to the ePython VM outlined here had to be smaller than 1.2KB in order to fit within the existing memory map and the Epiphany's SRAM constraints.

4.2.4 ePython communication primitives

Table 4.2 itemises the blocking communication primitives that underpin the pass by reference data transfer model discussed Section 3.2.2.1. The `value()` primitive requests the value of an external (host Python) variable and waits for the response (blocks). The `get_slice()` function requests a *slice* of an external list (vector) and allocates space on the micro-core to store it. The `set_slice()` updates the slice of the external list with the contents of the local micro-core list. The `get_slice_byref()` primitive requests a slice of an external list and updates an existing local micro-core list.

Table 4.2: ePython Xfer blocking communication primitives

Primitive	Description
value()	Returns the value of a host Python variable
get_slice()	Returns a new on-core slice of a host Python list
set_slice()	Updates a slice (or the whole) of the host Python list
get_slice_byref()	Returns a slice of host Python list into the provided buffer

The Xfer communications protocol and primitives are the same for both the VM and Olympus abstract machine versions of ePython, with only the underlying implementation of the primitives changed to reflect the different memory models. For example, whilst external variables in the interpreter are stored in the symbol table along with local variables, Olympus external variables are reference types stored in a special area of the heap. This allows the IDs for Olympus external variables to be encoded, along with a *dirty flag*, within their corresponding heap block header field. The dirty flag is used to determine whether or not the `getExternalVariable()` call is made to retrieve the data from the host. Therefore, the underlying communication primitives are implementation agnostic.

Table 4.3: ePython Xfer non-blocking communication primitives

Primitive	Description
async_get_slice()	Returns a handle to a request for a list slice
async_set_slice()	Updates a slice of a host Python list, returning a handle
value_byref()	Loads a slice of a host Python list into the provided buffer
wait()	Wait for the completion of an asynchronous transfer
ready()	Returns true if the asynchronous transfer has completed

The set of primitives listed in Table 4.3 support *non-blocking* requests, where communications and processing can be overlapped to prevent the kernels waiting on external variable data to be returned from the host. To support this, a new variable type *handle* was added to the ePython types, with non-blocking external data access primitives returning a handle containing a reference to the specific channel cell that was used to initiate the original request for external data. The `async_get_slice()` non-blocking primitive requests an external list slice from the host and returns a handle that is then used load the data when it is available. The `async_set_slice()` primitive sends a message to update an external list slice, returning a handle that can then be used later by the micro-core to confirm the update has been made on the host. The `value_by_ref()` primitive loads the slice of an external list into the local micro-core memory, using the handle previous created by a previous `async_get_slice()` call. The `wait()` primitive blocks execution until the handle confirms that the transfer is complete. A new `ready()` runtime primitive was added to test whether a non-blocking data transfer has completed by checking the status of the cell referenced by the handle variable. As shown in Figure 4.5, this can reduce the micro-core waiting time for *pre-fetched* external data requests to the amount of time it takes the micro-core to load the data from the channel cell(s), rather than the full amount of time from initiating the request to the host (shown as *on-demand* wait time).

Whilst the primitives are available to the ePython programmer, their direct use is not required as the `@offload` function decorator's parameters allow the programmer to control the automatic transfers, including the variables to be pre-fetched, the size of the local buffer on the micro-core to store the data and the amount fetched at a time, as detailed in Section 3.2.2.1.

4.2.5 Memory hierarchies and memkinds

The changes to support the host Python integration (Section 4.2.2), the new communication model (4.2.1) and ePython communication primitive functions (Section 4.2.4), coupled with a finer-grained selection of target location (local, shared or external memory) for variables and functions, provide the basis for the memory hierarchy and memkinds functionality (Section 3.2.2.2). Previously, ePython kernels and data could execute locally or from the shared memory area [101]. With memkinds, it is possible for the programmer to specify which level in the hierarchy a particular variable resides through the use of the *Host* (external), *Shared*, *Device* (other on-device

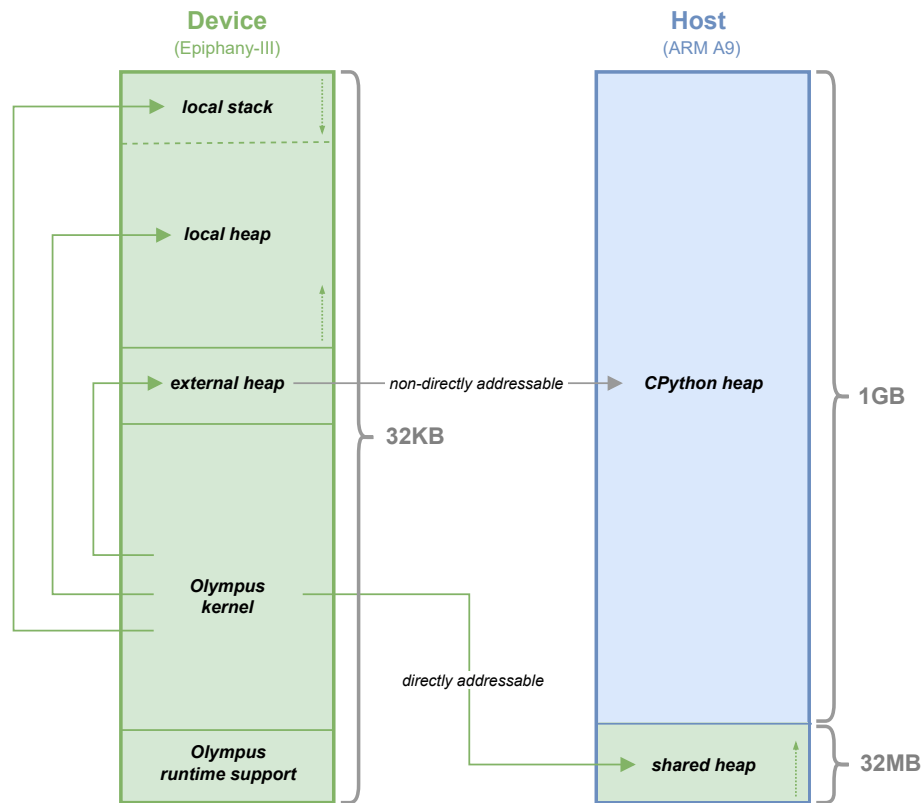


Figure 4.6: Memory map for the Olympus abstract machine on the Adapteva Parallella (not to scale)

core) and *Local* memkind classes described in Section 3.2.2.2. The memkind objects contain the type, location and number of elements to be allocated, and are underpinned by the heap allocator within the ePython runtime. For the VM, the heap allocator already supports creating variables within shared memory. For the Olympus abstract machine, as shown in Figure 4.6 for the Adapteva Parallella⁶, the heap was divided into areas for each of the memkinds: *local* core memory (Local memkind), *on-device* memory (Device), *shared* (Shared) memory and *external* (Host⁷) memory. The general kernel stack and heap objects are stored within the local memory area, shown in green at the top left of Figure 4.6. The on-device objects are directly addressed from the relevant heap of another core on the device (omitted from the diagram for clarity). The shared objects are directly referenced from a separate heap located within the shared memory area on the host, highlighted in green at the bottom right of Figure 4.6. The local copies of external objects are held within a separate area of on-chip memory below

⁶On the host, the Xfer channel / cell and ePython control area in shared memory, the CPython interpreter and the Linux memory map (kernel, drivers etc.) are not shown for clarity.

⁷Specifically, these are variables within the host CPython heap.

the local heap. As discussed in Section 4.2.3, transfer to and from the host memory can be implicit or via direct calls to the communication primitives.

The memory map for the Xilinx PYNQ MicroBlaze and RISC-V micro-core design is similar but with the memory available to the CPython heap reduced to 512MB [125]. Whilst it is possible to directly access the memory of other cores on the device, for both the Parallella and PYNQ micro-cores, within the Olympus abstract machine, this is not currently supported by the ePython VM.

Although GPUs and micro-cores have architectural similarities, as discussed in Section 2.2.1, the latter have more flexible programming models when they support memory hierarchies. Whilst the implementation of the pass by reference kernel data model required a number of changes including a C Python integration module, a new communications protocol between the micro-cores and host, and changes to the ePython device code, the benefits for the programmer are clear. By accessing the non-directly accessible 1GB of memory on the host, kernel codes can overcome both the 32KB on-chip and the 2MB per core shared memory restriction on the Adapteva Parallella. Furthermore, as this area is already used to support the communication channels / cells and kernel launch / control messaging, increasing the size or number of channels would put further pressure on this limited resource. The ability to support kernels running on the micro-cores that can manage data sets equivalent to even modest desktop machines, enables a much greater set of real world applications to be run on the supported micro-core devices.

4.3 Increasing kernel runtime performance (RQ3)

4.3.1 Native code generation options

Whilst implementations of dynamic languages, such as Python, for micro-core architectures greatly reduce the time and effort to develop applications in comparison to using the provided C software development kits (SDKs), there remains a significant runtime overhead of interpreting dynamic languages over compiled C binaries. There are two major approaches to accelerating Python codes: ahead-of-time (AOT) and just-in-time (JIT) compilation. AOT statically compiles the Python source code before it is executed and JIT dynamically profiles the bytecode⁸ as it is loaded at runtime, se-

⁸Although bytecode is a specialisation of a portable code format, namely a single-byte opcode, and the terms are often used interchangeably, in this work *opcode* will refer to the instructions themselves

lecting the most appropriate sections of code (*hotspots*) for native code generation. AOT compilation is usually performed on the host, leveraging the significantly greater amounts of memory than is available on the device but it can also be performed on the device where all bytecodes are translated to native code before execution (MicroPython) [126]. JIT compilation, by its nature, has to be performed on the device executing the bytecode.

As discussed in Section 2.5, the ePython VM only requires 24KB of on-chip memory, including the interpreter, runtime support and communication libraries, which is significantly smaller than the 256KB required by MicroPython [85]. However, MicroPython provides the *viper* AOT native code compiler that increases code runtime performance by a factor of seven [126] over the interpreter. Here, the key question is whether AOT or JIT compilation is the most applicable on micro-core architectures. Furthermore, for both approaches, the design of the *virtual machine*⁹ [127] has an impact on runtime performance and ease of implementation, where the different models for opcode addressing will impact runtime performance relative to code generation complexity [128]. This is important for a JIT compiler as it profiles the executing opcodes and then selects targets for optimisation, with the remainder of the opcodes being executed by the interpreter. Therefore, it is of value to improve the runtime performance of the existing virtual machine itself and understand the trade-offs of the different options. For example, whilst a register-based virtual machine would be expected to outperform a stack-based alternative, as is the case for the Lua programming language [74], the code generation complexity for a register-based virtual machine is significantly greater due to operations such as register allocation [129]. Furthermore, different opcode addressing models can impact the size of the opcode binaries and any conflicts between code size, runtime performance and compiler complexity need to be explored.

4.3.2 The Merlin virtual machine

The runtime performance of different opcode dispatch models was also of interest, bearing in mind that certain options, such as *computed gotos* (GCC *labels as values*) [130], are not available across all compilers. Therefore, as part of designing a

and *bytecode* refers to the file containing them.

⁹Here, the definition of virtual machine comprises the opcodes and the interpreter. The two are interlinked as the opcodes are executed by the interpreter but the underlying implementation is not necessarily coupled to the design of the opcodes, for example, different opcode dispatch models.

JIT compiler for ePython, the Merlin virtual machine¹⁰ [131] was designed and built to support the comparison of *stack-based*, *three-address* and *register* opcode modes, as well as the comparison of *switch table*, *function pointer* and *computed goto* dispatch modes. Furthermore, Merlin supported a review the existing ePython bytecode and virtual machine design to determine if improvements to bytecode size and runtime performance could be obtained. The Merlin virtual machine was also designed to be portable across different processor architectures (including Epiphany-III, MicroBlaze, RISC-V and SPARC) and C compilers (including GCC, Clang, SDCC and CodeWarrior), to determine any impact on the implementation of a new virtual machine due to the features of different CPU architectures. As part of the project's portability objective (RQ4), processor architectures with different byte ordering, such as the Epiphany-III and SPARC, were chosen to expose, and remediate, the issues of developing a cross-platform virtual machine with a common bytecode. Furthermore, the byte alignment requirements of the Epiphany-III and SPARC impact the virtual machine's memory model including the layout / padding of elements on the stack and the block structure of the heap manager, making them ideal targets for the design of a new virtual machine.

Merlin makes extensive use of C macros, as shown in Listing 4.2, to abstract the opcode dispatch model from the individual opcode declarations and underlying processor architectural differences. This enables the reuse of opcode definitions across different dispatch models (controlled by a compiler flag), and supports the implementation of multiple opcode addressing models at the same time. The `OPCODE_BEGIN` and `OPCODE_END` macros delineate the opcode definitions and abstract the underlying dispatch mechanism. The macro definitions for the different opcode dispatch models are listed in Appendix D.

¹⁰Merlin was not based on the existing ePython VM as the ePython bytecode is a flattened, tokenised parse tree, with symbol table references, thereby precluding the assessment of different addressing modes on bytecode size and runtime performance.

```

1  OPCODE_BEGIN (ADD)                /* Stack-based */
2      DEBUG_OUT (ADD) ;
3      tmp1 = POP;
4      tmp2 = POP;
5      PUSH (tmp1 + tmp2);
6      NEXT;
7  OPCODE_END
8
9  OPCODE_BEGIN (ADDM)               /* Three-address */
10     DEBUG_OUT (ADDM) ;
11     GET_ADDRESS (tmp1) ;
12     GET_ADDRESS (tmp2) ;
13     GET_ADDRESS (tmp3) ;
14     ACC = ILOAD (tmp2) + ILOAD (tmp3) ;
15     ISTORE (tmp1, ACC) ;
16     NEXT;
17 OPCODE_END
18
19 OPCODE_BEGIN (ADDXY)              /* Register-based */
20     DEBUG_OUT (ADDXY) ;
21     ACC = X + Y;
22     NEXT;
23 OPCODE_END

```

Listing 4.2: Example Merlin opcode definitions

The Merlin bytecode is little-endian to match the majority of target processors but support for big-endian processors, such as the SPARC and PowerPC, is provided by swapping the operand byte ordering as the opcodes are loaded on these targets. Macro definitions manage data and address sizes, byte ordering and alignment, enabling the construction of virtual machines with differing data width and address size; for example a 32-bit data width with 16-bit addressing, suitable for the 32KB and 64KB Epiphany, MicroBlaze and RISC-V micro-cores. In Merlin, like the Pascal *p-machine* [132], memory addresses are offsets within a frame rather than memory locations and the virtual machine contains opcodes for block / function entry (ENTER / LEAVE), scope management (GLOBAL, LOCAL and SCOPE) and scoped memory access opcodes (e.g. local LDI / STI and global LDGI / STGI). A *cross overlay* file for the Python-based SB-Assembler [133] was created to enable tests to be written in an assembly language for the Merlin virtual machine, greatly simplifying the creation of

codes to investigate the trade-offs discussed in Section 4.3. The runtime performance and bytecode size comparisons of the different opcode addressing models are presented and discussed in Section 6.3.1.

4.3.3 Olympus compiler and abstract machine

A key driver for the design of either a JIT or AOT compiler is not only to improve runtime performance, whilst ideally reducing the memory footprint, but also to maintain the high-level programming features of dynamic programming languages, such as *closures* [134] and dynamic typing. These features introduce runtime complexities that require careful implementation to avoid excessive runtime performance overheads. For example, managing dynamic types by *boxing* and *unboxing* the values introduces significant marshalling overheads for each reference / update [116], or the runtime performance impact of managing free variable references in closures by chaining up the *environment* [135] links. Over and above the requirements that dynamic languages place on the approach to native code generation, a key requirement in designing a new AOT compiler for ePython was to retain as much of the existing runtime support as possible, as it provides mature parallel programming features, with the required underlying communications support. The main components of the ePython environment were isolated: the compiler, interpreter, communications and runtime code. Whilst the interpreter would not be required in the new design, elements such as the symbol table, and associated data structures, are used as part of the ePython runtime support and were carried across to the new native code generation environment.

At a high level, the native code generation approach employed in this work is similar to that of the MicroPython *viper* emitter in that native code is generated for *all* Python source code, rather than leveraging interpreter functions for the execution of arithmetic and comparison operations. However, in this work, C source code is generated that is compiled to a native binary for download and execution on a micro-core device. A number of existing programming languages have used this approach for traditional computing environments, including Eiffel [136], Haskell [137] and Lua (Palene / Titan AOT compiler) [138]. Whilst Haskell has deprecated the C backend in preference to a new LLVM [94] backend, the former is still beneficial for porting to a new platform as it produces *vanilla code*, requiring only GCC, `as` and `ld` tools [139]. Likewise, C was chosen as the backend for the ePython AOT compiler to enhance portability, particularly as a number of the target micro-core architectures, including

the Adapteva Epiphany and Xilinx MicroBlaze, are not supported by LLVM¹¹. Furthermore, code generators, such as MicroPython’s emitters, need to be specifically written to support new processor instruction set architectures (ISAs), and the C back-end ensures that native code generation is immediately available on all micro-core platforms that ePython supports. Crucially, the approach adopted here generates high-level C source code that retains the overall structure of the Python source program rather than emitting machine code representations of the bytecode (MicroPython) or disassembling / translating the bytecode to native code (Numba). The C source code generation approach was adopted in order to leverage the extensive optimisation capabilities within modern C compilers, including register allocation, data flow analysis, instruction selection and scheduling, data dependency management and scalar optimisations [129].

4.3.3.1 Overview

The new Olympus abstract machine was designed to meet the requirements for the AOT compilation of dynamic languages, such as ePython, and leverages the experience gained developing the Merlin virtual machine, specifically its memory model. The Olympus approach describes a high-level abstract machine in C that supports dynamic programming languages and provides a simple target for the source language (ePython) compiler. This approach of generating C code, with associated abstract machine *mnemonics*¹², is informed by the Anubis¹³ abstract machine and is similar to the cross-platform, macro-based, code generation approach used by [143]. However, the Olympus abstract machine addressing model, based on that for Merlin, is static and generated by the compiler, whereas the Anubis abstract machine performs dynamic chaining through the environment to lookup objects, in a similar manner to the searching of the symbol table in the ePython virtual machine. Furthermore, the Olympus model separates object addressing from the operation mnemonics, enabling the support for all areas of micro-core memory hierarchies and the placement of objects on the stack or in the heap, as required. The Olympus memory addressing model is de-

¹¹At the start of this research, only GCC supported the RISC-V.

¹²Macros or functions that represent abstract machine instructions e.g. `DECLR` to declare a real object, `STI` to store an integer, `ADDRL` to address a local object and `MKLAMBDA` to create the structure containing a λ and its static environment. Appendix F contains a listing of all of the Olympus mnemonics.

¹³Anubis [140] is a persistent [141] Object-Oriented programming language that contains powerful programming abstractions, such as *first-class procedures* [142] and *anonymous functions* (λ s) [23], with their respective closure support.

scribed in Section 4.3.3.2. The mnemonic model allows code generation complexity to be managed in the compiler or abstract machine, as is expedient. For example, the instructions required to copy a function return value before the stack frame is *retracted* could be generated individually by the source language compiler or they could be implemented as steps within the function `APPLY` and `RET` mnemonics. Moreover, if the complexity is hidden within the abstract machine mnemonics, the code generation routines within the compiler do not need to be changed if the decision is made at a later date to manage source function return values using C’s in-built function call mechanisms. Currently, the mnemonic model allows Olympus to manage cross-compiler issues. For example, the different function call order of GCC and Clang is managed by the `EXPR` mnemonic that performs no operation for GCC but stacks the return values of expression mnemonics containing multiple function calls to ensure the correct evaluation order for Clang. The Olympus code generator does not need to be aware of the C compiler being used to compile the generated abstract machine mnemonics, it only has to emit a wrapping `EXPR` mnemonic for expressions containing multiple function calls.

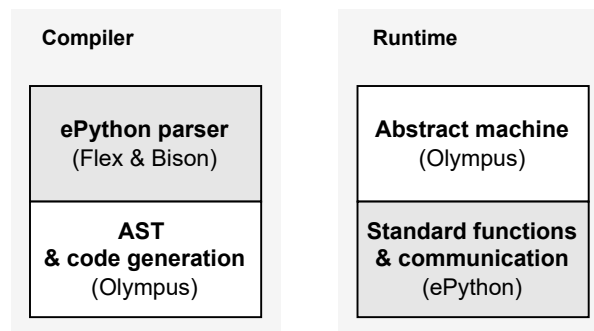


Figure 4.7: New Olympus and existing ePython components

The Olympus codebase includes functionality to support code generation for the abstract machine, including a new AST library. Figure 4.7 outlines the new Olympus and existing ePython components (highlighted in grey) that comprise the new ePython native code generation framework.

Figure 4.8 shows the Olympus code generation phases for a dynamic language such as ePython. For ease of development and testing, the phases are *pipelined* i.e. the module to generate the AST in Phase 1 is separate from that which traverses the AST to generate the Olympus abstract machine code in Phase 4 and the output of one phase (the AST) can be *piped* into another. This also allows the AST to be dumped as a textual representation, or to generate a graph using Graphviz [144]. The textual

AST files enabled the testing of individual AST processing phases 2-4 (type inference, basic optimisation and code generation), without having to run the code end-to-end. A downside of this approach was the impact to both the AST writing / parsing code when structural changes were made to the AST, increasing development effort. However, this was offset by the reduction in the time taken to fault-find code generation issues, as they could be isolated phase by phase where the textual representation of an AST of previously compiled ePython codes from Phase 1 could be used as the test input for each phase and the visual representation (Graphviz) of the output AST inspected.

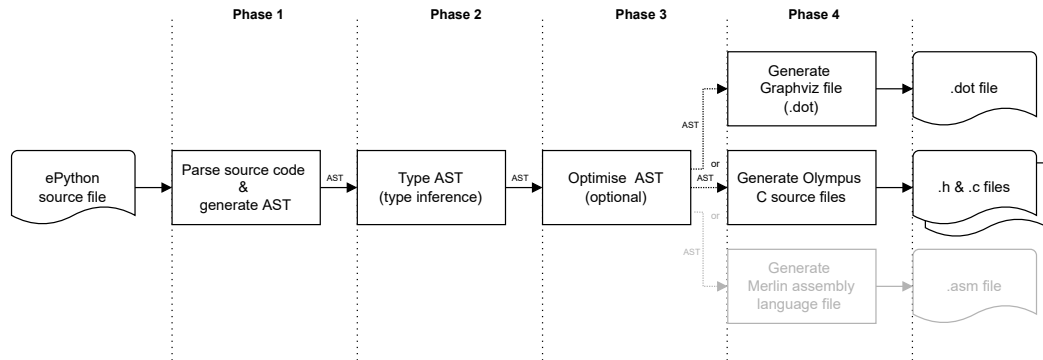


Figure 4.8: Olympus code generation phases

As shown in Figure 4.8, the code generation phase (Phase 4) can support different *pluggable* code generator *targets*, for example, Olympus abstract machine C source code and Merlin assembly language files (`.asm`), thereby allowing different target code to be generated from the same AST. Two code generators have been fully developed to support the research aims of the project: one for the Olympus C abstract machine and another for the Graphviz `.dot` file format to allow the structure of the AST to be inspected. The feasibility of pluggable executable code generators was tested by creating a code generator for a subset of the Merlin assembly language. The key driver for the pluggable code generator target API was to enable the future implementation of an ePython bytecode target to allow the replacement of the existing ePython monolithic compiler and the future implementation of hybrid Olympus native code / bytecode kernels.

4.3.3.2 Memory model

The main departure of the Olympus abstract machine from simple *transpiled*¹⁴ code is that the objects in the target code are not translated to C variables but to elements within *environments* containing *frames*, as shown in Figure 4.9. Therefore, all memory management is managed by the abstract machine, not the C compiler and runtime; this includes function stacks, argument passing and return values. Furthermore, as `libc`, the C runtime library is often too large to be used on many micro-core architectures, the abstract machine also provides a simple, compact heap manager.

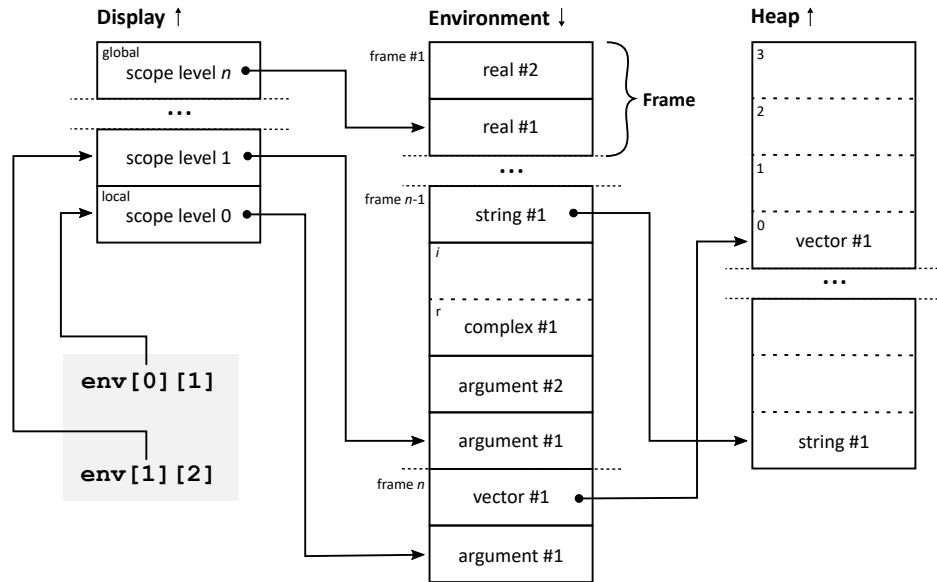


Figure 4.9: Display, environment and frame structure

The generated code declares all objects within frames as shown in Figure 4.9, that grow downwards from the top of memory. Following [145], a *display* [146] is created that holds references to the frames, allowing object indexing via scope (*lexical*) level and offset. Olympus objects are *lexically* scoped by the compiler based on their nested scope level, rather than dynamically scoped at runtime. The scope level is inverse to the lexical level i.e. referencing an object in the enclosing scope *increases* the scope level, and the scope level and offset are the *static-distance coordinate* of the object [129]. The compiler calculates and maintains the levels and offsets of the objects within the environment, increasing runtime performance as the addressing is performed statically within the compiler, rather than at runtime by chaining up the environment. A new frame is created for each function call (β reduction) and the display in the abstract

¹⁴A form of source-to-source compilation where objects in the source language code are transformed into equivalents in the target language.

machine is updated to manage the access to objects in outer frames (enclosing scope). For most function calls, a new entry is created in the display but recursive calls reuse the same display entry updated to point to the new frame for each invocation. This allows all objects to be referenced by scope level and offset for recursive functions, thereby maintaining the ability to access all objects by indexing from a base pointer, negating the requirement to chain up the environment list to find objects in outer scope levels.

Figure 4.9 also shows that all fixed-size objects can be allocated in a frame or in the heap. Here, a complex number is declared in frame $n-1$, rather in the heap as for the vector in frame n . Allocating space for objects in the frame is much quicker than allocating them in the heap as the process of memory allocation is simpler, where the frame pointer is just decremented by the required amount. This model allows decisions to be made within the compiler regarding the best placement for composite data types.

A more detailed explanation of the management of environments can be found in [146], and [135] discusses the issues with simple displays for environment and closure support. These are standard techniques for the implementation of a static *block-structured* programming language, such as C, Pascal or Java. For the Olympus abstract machine, these mechanisms are being used to support type freezing (discussed in Section 3.4.1.1) for a dynamic programming language, such as Python. Effectively, the dynamic typing of Python is being translated, or frozen, into a static typing model that can be supported by traditional static block-structured language implementation techniques¹⁵. Similar to [135], an *environment link* is created within the frames that is used to support the copying of frames to the heap for functions that return objects within the closure to enable them to be referenced once the frame is retracted. Without this copy and update of the display to point to the closure now stored in the heap, these references would be *dangling* after retraction. This is the approach taken by Lua [74] and was chosen for Olympus as the vast majority of kernels executing on the micro-cores do not use closures. Furthermore, as the allocation and deallocation mechanisms for frames stored on the stack are much more efficient than for those stored within the heap, it was deemed preferable to impact these less common closure-utilising codes in preference to impacting all codes, including general scientific kernels.

¹⁵It is important to note that full block-structure support (environments) is required to support Python features such as lambdas, rather than the more limited block-structured model of C.

4.3.3.3 Type management

All target object declarations are typed, either via type inference performed on the AST for the vast majority of codes or directly from type annotations¹⁶ in the source code. Furthermore, all object access is similarly typed, with dynamic changes of an object type resulting in a new typed declaration and updated typed access from that point of execution onward. This is managed by creating a new `ASTdeclaration` and associated `ASTidentifier` node within the AST with the updated type. It is important to note that this does not require the abstract machine to allocate any additional space within the stack frame for the new object as, by default, *compound types* (arrays and complex numbers) are allocated in the heap, with a reference to the new object replacing the previous value stored within the frame. This model removes the requirement to perform dynamic runtime type checking and this approach to typing is similar to the simple type freezing performed by [113] in that runtime type checks are not performed on objects, thereby eliminating the associated overhead and increasing runtime performance.

4.3.3.4 Memory management

Listing 4.3 is an example of the typed object access in the generated C code for the memory layout visualised in Figure 4.9. In this example, the real part of the `complex` object at the enclosing (*foreign*) scope level 1 and offset 2 (`ADDRF (1, 2)`) is being set to the value `4.3` by the store complex real (`STCR`) mnemonic. The second line updates the integer element indexed by the object at offset 0 (`LDI (ADDRL (0))`) of a vector stored in the heap and declared in the local scope at offset 1 (`ADDRL (1)`), the value `42`.

```

1 STCR (ADDRF (1, 2), 4.3);
2 STAI (ADDRL (1), LDI (ADDRL (0)), 42);
3 STR (ADDRF (4, 1), 10.0);

```

Listing 4.3: Object access example

Figure 4.9 also highlights that the scope level increases outwards from the local block (or function) scope, with local objects having a scope level of zero. This enables the C compiler to use indexed addressing from the frame pointer (`env`) to access local

¹⁶In the case of ePython, these are the type constructor functions `int()`, `float()` and `str()` which can be considered similar to the casting operators in C, and can be used to help the compiler infer types when it returns an error that it is unable to do so.

objects, thereby increasing runtime performance for local objects and loop block indices, as the compiler can use indexed addressing to directly access the local objects. Furthermore, the indexing of outer scope levels directly removes the need to chain up environment frames to locate non-local objects, with the corresponding runtime performance overhead. This model also allows support of Python 3 `nonlocal` objects [147] that are declared in the nearest enclosed scope level.

In the generated C code, the object access APIs are implemented as mnemonics that directly update the frame elements within the environment and include the required casting to and from the current object type. This is illustrated at line 3 of Listing 4.3, where the `STR` and `ADDRF` mnemonics combined expand to `((*(Real*) (&((Value*) memory->senv[(level)])[(offset)]))=(Real)(value))`. These not only access the frame element directly, but also ensure that the value is stored correctly in memory. As in this example, casting a real value to and from the generic `Value` field within the static environment (`memory->senv`). Whilst, conceptually, Olympus utilises an environment link described by [135], the implementation is different. The approach here is to store the current display frame (`memory->frame`) as the λ 's static environment within the `Lambda` structure created by the `MKLAMBDA` mnemonic at the time of declaration. This allows the Olympus abstract machine to support first-class and nested functions.

The variable referencing mnemonics of the initial Olympus memory referencing model performed the addressing of objects within the frames directly. However, in order to support multiple levels in the memory hierarchy and compound types such as complex numbers on the stack, as well as in the heap, the separation of addressing from access within the mnemonics was required. The new model also simplified the implementation of object references (the `&` operator in ePython and the `id()` function in Python) within the abstract machine as the `ADDRL` and `ADDRF` mnemonics can provide the object address within the stack or the heap. The implementation of object references and the unified `Memory` structure discussed in Section 4.3.3.5, enables the integration of Olympus applications with C frameworks, such as the Eithne benchmarking framework [25] and MPI (Message Passing Interface) [48].

4.3.3.5 Function stack frame management

```

1 void oly_i$add$ii(Memory *memory, Object *self) {
2     STI(ADDR(0), (LDI(ADDR(1)) + LDI(ADDR(2))));
3 }

```

Listing 4.4: Generated function example

As shown in Listing 4.4, all generated Olympus functions are passed two parameters, a pointer to the current `Memory` structure¹⁷ and the current object pointer (`self`). As detailed in Listing 4.4, the `memory` structure contains the dynamic environment of frames (`memory→display`), the current frame (`memory→frame`), the static environment of frames (`memory→senv`), and pointers to the heap, the expression value and function return stacks. The `self` function parameter is a reference to an `Object` type which allows the abstract machine to support Object-Oriented, where a function is a method (*member function*) of an object, similar to the implicit `this` pointer in non-static member functions in C++ [148] and the `self` parameter in CPython methods [149].

```

1 typedef struct memory {
2     Word size;                                /* Heap size */
3     Byte *top, *limit, *base, *free;          /* Heap pointers */
4     Frame tos, *display, *dbase; /* Stack and display pointers */
5     Frame frame, *senv; /* Current frame & static environment */
6     Value *vs, *rs; /* Value and return stack pointers */
7 } Memory;

```

Listing 4.5: Olympus memory structure

Listing 4.4 illustrates that Olympus functions do not use the C function stack frames for source language (ePython) function parameters or return values. Therefore, the function parameters and return value are declared within the same frame as the function’s local objects and form part of the overall frame size, managed by the Olympus abstract machine, as shown in Figure 4.10. The return value is at offset 0 within the frame, with the parameters starting at offset 1 and local objects following

¹⁷The original Olympus function call interface discussed in [27] passed the environments individually, using global pointers for the heap, expression value and return stacks. However, not only did this approach complicate integration with C frameworks when using Olympus functions as *callbacks* but it was also incompatible with support for multithreaded Olympus functions, necessitating the `Memory` structure approach discussed here.

the last parameter. For example, if there were any local objects declared in the `add()` function in Listing 4.4, the index of the first variable would be equal to 3.

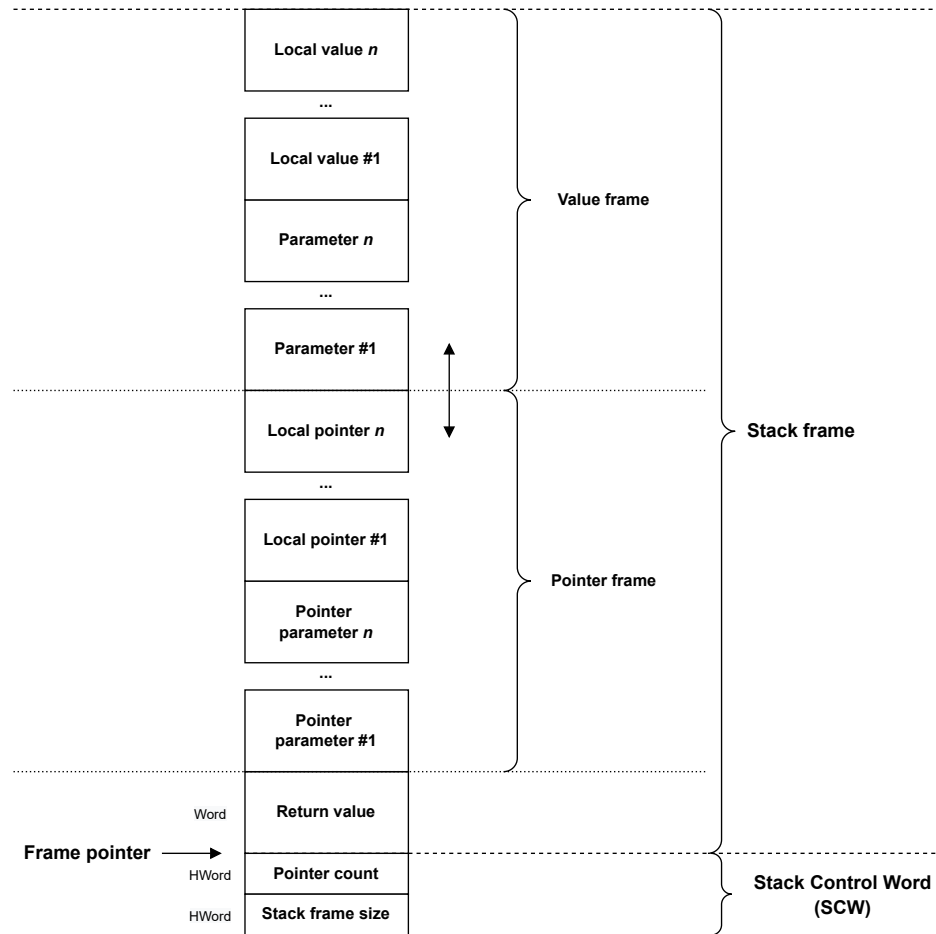


Figure 4.10: Olympus stack frame structure

Function parameters or local objects that are pointers are placed in the *pointer frame* at the beginning of the stack frame, just after the return value, and all other non-pointer function objects are placed afterwards in the *value frame*. The placement (update of offsets) is performed by the compiler, with the Olympus abstract machine mnemonics for pointer and non-pointer object access remaining unchanged. This layout¹⁸, coupled with the *pointer count* field in the *Stack Control Word* (SCW), allows the deliberately simple heap compactor to quickly update stack pointer objects when heap blocks are moved down in memory to reclaim space. This design was chosen due to its low memory footprint, both in terms of heap pointer management structures and

¹⁸The use of separate pointer and non-pointer stack frame areas is similar to the separate pointer and main stacks in S-algol [150] except that, for Olympus, the areas are within the same frame, simplifying the abstract machine access mnemonics.

the heap compaction algorithm code. Furthermore, this design removes the requirement to implement *double-indirect* pointers [151] for heap objects. This increases object access performance, particularly for in-heap arrays and complex numbers, at the expense of additional complexity in the compiler to manage the placement of function parameters and local objects into the correct frame area, rather than trivial offsets from frame pointer. However, the complexity, including the movement from one stack area to another if an object type changes from a non-pointer to a pointer type and vice versa, is constrained within the compiler code generation phase using an AST *operation*¹⁹ node that updates the object's offset, thereby placing it in the correct stack area, and then updating all references using the object's unique AST ID field. The same process allows Python *keyword* (named) parameters, as shown in Listing 4.6, to be supported by the abstract machine, where the parameter names are used to map the offsets. Fundamentally, the mapping of offsets is managed by the compiler at compile time, rather than at runtime by the Olympus abstract machine and the simple offset-based addressing model remains unchanged, with its associated memory footprint and runtime performance benefits.

```

1 maurice@vipera:~/build/scratch$ cat parampos.py
2 def fn(a,b,c):
3     print(a + b + c)
4
5 fn(b="b", c="c", a="a")
6
7 maurice@vipera:~/build/scratch$ python3 parampos.py
8 abc
9 maurice@vipera:~/build/scratch$
```

Listing 4.6: Python keyword parameter example

The `Memory` structure model not only simplifies the frame management (*stack retraction*) but also provides the capability to support the removal of C function bodies for static (non-dynamically loaded) Olympus functions and allow the implementation to use C jumps to labels (*gotos*) or GCC *labels as values* [152] or even enable the *inlining* [148] of Olympus functions. Crucially, by leveraging frames within an environment, the decoupling of objects from the underlying C storage mechanisms not

¹⁹AST operation nodes allow functions to be called dynamically as the AST is processed. Here, it is required as the update of the reference offsets cannot be static (when the object update AST node is created or a separate pass) as this would break the previous object references.

only provides support for the dynamic and Object-Oriented features of many source languages, but also for the implementation of dynamically loaded functions, which is described in Section 4.4.

4.3.3.6 Code management

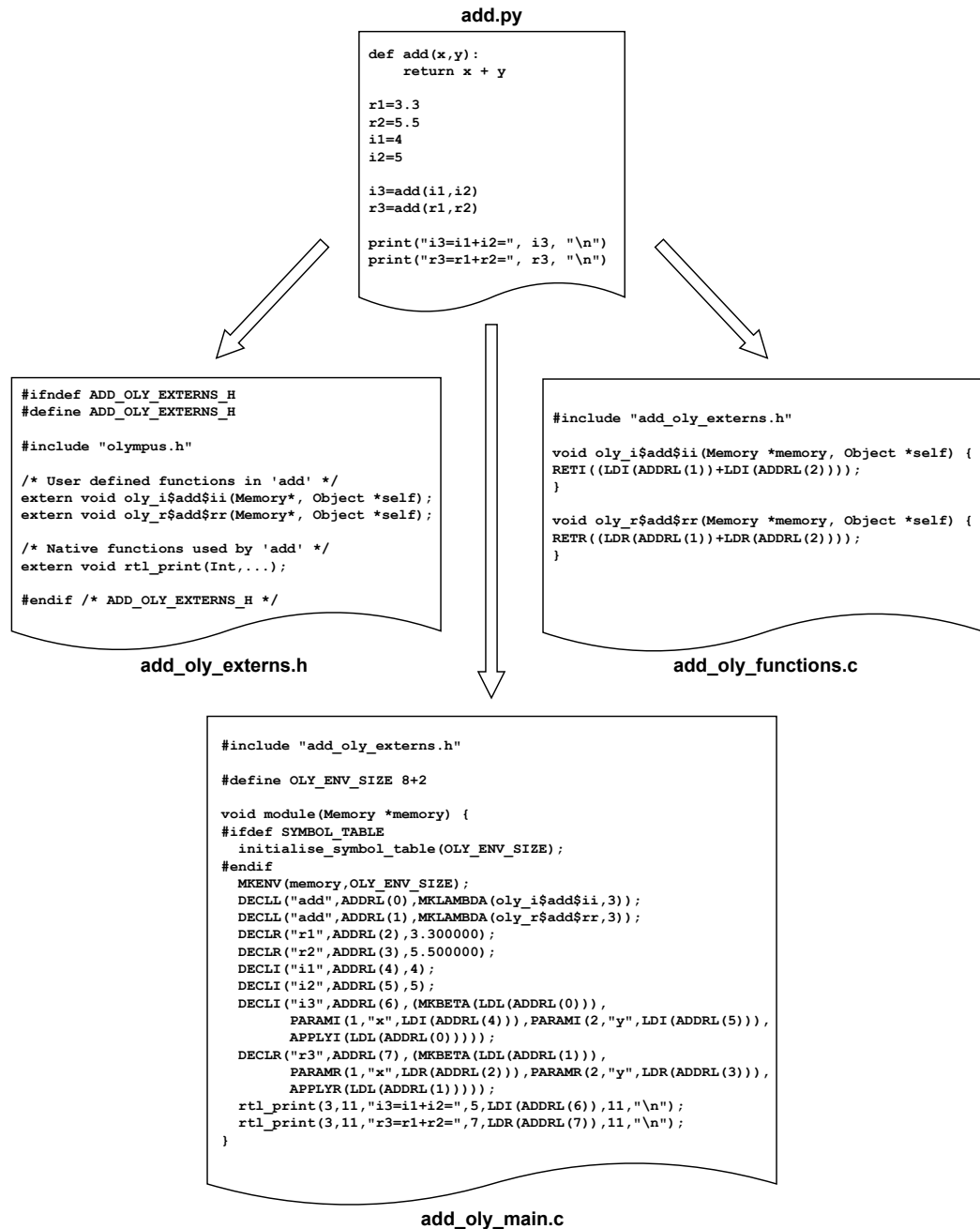


Figure 4.11: Olympus generated file structure

Figure 4.11 presents the structure and contents of the files generated for a simple ePython example (`add.py`). For all source language files, the Olympus code generator (`olygen`) creates three C source files:

- A `<filename>_oly_main.c` file containing the main body of the source file
- A `<filename>_oly_functions.c` file containing all of the Olympus generated functions from the source file
- A `<filename>_oly_externs.h` file containing all the C function references, including the Olympus generated functions and standard functions provided by the Olympus abstract machine runtime

Where `<filename>` is the source language filename less the extension (e.g. `.py`).

The separation of the generated functions from the main body of the source file enables the support of dynamic function loading, as well as separate compilation and object linking. For the latter, the `_oly_main.o` object file can be discarded for programs that link to the functions in the `_oly_functions.o` object file (referenced by the `_oly_externs.h` header file). In the same manner, for the former, the `_oly_functions.o` object files are interrogated and loaded into memory on the device by the dynamic function loader. The original implementation of dynamically loaded functions only separated out the dynamic functions from the main body file but the approach was widened for all functions to simplify the implementation, whilst providing support for separate compilation, which greatly simplifies the development and debugging of non-trivial programs. The ePython source and generated Olympus abstract machine files for the Sieve of Eratosthenes benchmark are listed in Appendix C.

The Olympus code generation model was specifically designed to address runtime performance and memory constraint concerns. Furthermore, the design needed to be flexible to be able to support the peculiarities of differing micro-core architectures. As shown in Chapter 6, the approach is widely portable to a number of different processor architectures, whilst also generating compact and fast code, that not only matches the runtime performance of native C for the Fibonacci and LINPACK benchmarks (Section 6.5.2) but also results in a significantly increased amount of precious on-chip memory available for kernel data required for scientific codes.

4.4 Managing arbitrary code size (RQ2)

As discussed in Section 4.2, on-chip SRAM is a precious resource on micro-core devices and must be carefully managed in order to run complex codes. What is true for kernel data is also true for kernel code and techniques need to be developed to enable the execution of arbitrary sized native code kernels. There are two aspects to managing this on micro-core architectures:

1. Reducing the footprint of compiled kernels and associated runtime support, for example math libraries.
2. Enabling dynamic loading and unloading of binary code to allow the execution of codes larger than the on-chip SRAM.

To achieve the first aspect, the code generation mechanisms need to be designed to produce as compact code as is possible and this can be aided by selecting C compiler options that reduce the generated binary size. However, the target codes are scientific kernels, there is a trade-off between code size and runtime performance. Judicious selection of individual low-level C compiler options will provide the optimal balance for specific kernels and runtime functions. However, the focus of this work is developing techniques to support the second aspect, namely mechanisms to overcome the on-chip memory restriction via dynamic code loading and management. With this in mind, an investigation to find the best compiler optimisation flag combinations for the test kernels and runtime support functions was not undertaken and the code was compiled with the GNU GCC `-O3` and `-Os` compiler options.

As the ePython VM requires approximately 24KB of on-chip memory, when compiled with `-Os`, this determines the minimum footprint of deployed kernels. The ePython VM could be modified to dynamically load and unload functionality, such as the garbage collector and native (*standard*) functions, to reduce the minimum footprint of the VM. However, this would require significant structural re-engineering of the VM, including the implementation of a dynamic loader for native C functions. This loader would need to resolve all external symbols within the dynamically loaded C function, necessitating the chaining through the compiled Executable and Linkable Format (ELF) [153] object files to ensure all external references are marked and loaded. In other words, a dynamic loader for C functions would have to dynamically load all the symbols and functions it references and then the symbols and functions that any of these functions reference, and so on. The limited on-chip RAM precludes performing

this resolution on the device as all binaries are *stripped* [154] prior to downloading to the device to save memory. Whilst the problem of symbol resolution remains for dynamically loaded Python functions, we have much greater control over the process and can introduce features within the Python compiler to help simplify the implementation. For example, the dynamic functions in the Python source code can be annotated to help the compiler manage the dynamic linking process by marking these functions and compiling them separately from the rest of the kernel code. The next section will detail how the abstract machine and memory model discussed in Chapter 3 can be leveraged to support the dynamic loading of natively-compiled Python kernels.

4.4.1 Olympus dynamic function loading

For ePython kernels, the `@dynamic` decorator was introduced to mark functions for dynamic loading on the device, as shown in Listing 3.5 in Chapter 3. When `defer` is set to `False` or is unset, as for the `add_nums` function, the compiler generates the code in Listing 4.7. The function body code is replaced by the `DYNLDF` API call, which initiates the download of the `add_nums` function from the host, allocates the space in the device's heap to hold the function code and declares it within the current frame at offset 1. This is the only modification required to the generated code C to support dynamic loading, the existing lookup and function application mechanisms remain unchanged.

```
DECLL (1, "add_nums", DYNLDF ("oly_i$add_nums$ii", 1)) ;
```

Listing 4.7: Dynamically loaded function declaration example

The `DYNLDF` API call takes the mangled name of the ePython function that is used to lookup the correct compiled code variant of the original function in the on-host symbol table and loader, based on the applied function's argument and return types. If multiple variants of the function are applied in the code, the corresponding variant declaration statements and associated dynamic loading calls will automatically be generated by the compiler. The API is also passed the total count of the function's parameters and local variables to enable the runtime to allocate a frame of the correct size on the device. Apart from the function code residing in the heap, rather than the binary code segment, the execution is the same as dynamically dispatched, statically loaded functions. The runtime performance and memory implications of the two function loading and dispatch models will be discussed in Chapter 6.

```

1 DECLL (0, "add", NULL) ;
2 ...
3 STL (ADDRL (0) , DYNLDF ("only_i$add$i", 2)) ;

```

Listing 4.8: Deferred dynamically loaded function example

Listing 3.5 illustrated how ePython leverages Python’s first-class function support to enable the deferment of dynamic function loading, using the `@dynamic(defer=True)` decorator and mark the function `add` for deletion from the heap after it has been executed. Listing 4.8 shows the code that is generated for these dynamic functions where loading is deferred to a later point of execution after declaration. This allows the programmer to control the exact time during a kernel’s execution a function is loaded and marked for deletion. The model allows the programmer a large amount of flexibility, with the ability to choose whether functions are statically bound to the binary that is downloaded to the device or to download the function at runtime, either when declared or just before execution and retain or delete them from the heap as the kernel execution profile demands.

The ability to separate the loading of a dynamic function from the declaration also allows the ePython compiler to implement a dynamic code loading strategy tuned to a kernel’s particular execution profile. Furthermore, as the code for dynamic functions is stored within the abstract machine heap, it can be discarded (*freed*) as required, thereby allowing the execution of much larger kernels than is possible with previous static code loading model.

The key hypothesis is that kernels with dynamically loaded functions can significantly reduce the memory requirements and enable codes of arbitrary size to be executed. Crucially, these codes are significantly larger than the kilobytes of memory available on micro-core devices. At a minimum, the model described here need only load a resident bootstrapper which contains the core support for marshalling and control of dynamically loaded functions. These can then be retrieved on-demand and marked for removal then garbage collected as the heap becomes exhausted for other objects.

4.4.2 Implementing the dynamic loader support

Figure 4.12 outlines the key components of the dynamic loader when applied to ePython. The updates to support dynamic code loading in the abstract machine were relatively

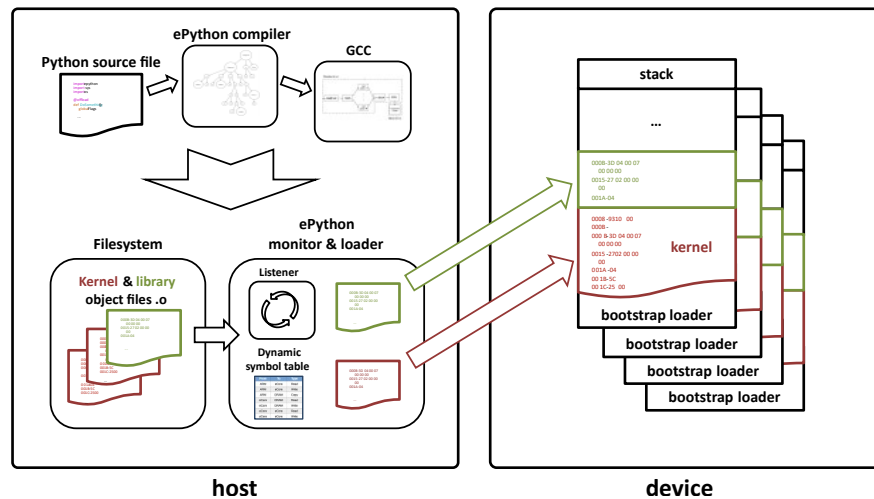


Figure 4.12: ePython dynamic loader architecture

minor; as the memory model already provided the abstraction of the variables and functions from the underlying C runtime, the modifications were mainly concerned with the request / transfer of the dynamic functions from the host and their loading into the heap. However, the changes to the host-based compiler and device support functions were more significant, requiring changes to the compiler, the Python integration module and monitor, and a new object file parser.

There are effectively two options available for dynamic loading; (1) make all user functions dynamic or (2) allow the programmer to select which functions they would like to be dynamically loaded. Initially, the former approach was chosen for ease of implementation but for increased flexibility the latter was introduced to allow the programmer to annotate dynamic functions. During the traversal of the Python AST during code generation, functions that are annotated with the `@dynamic` decorator, as shown in Listing 3.5, are placed in a separate C source file, with another file containing the bootstrap loader and dynamic loading calls for the relevant functions. A dynamic function symbol table is generated by the compiler and loaded by ePython on the host, which is then used to map the kernel dynamic function requests to the correct object file, highlighted in green in Figure 4.12.

The GCC C compiler is used to generate code for the target platforms (Epiphany-III, MicroBlaze, RISC-V, SPARC, MIPS32 and x86), resulting in ELF object files. Figure 4.13 illustrates the ELF file structure, highlighting the linkage between the different sections that need to be traversed to access the required function binary code. The dynamically loaded functions are keyed by their name and the host-side symbol

table provides the mapping between the ePython source view and generated C function names.

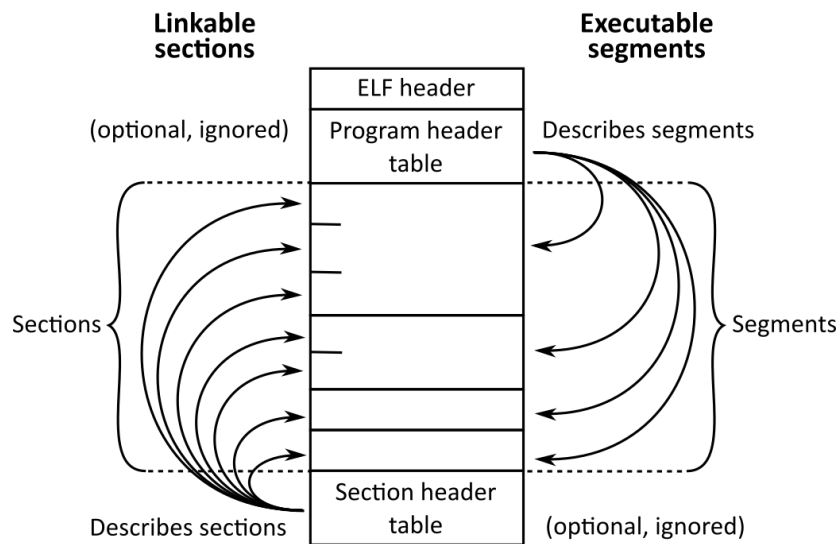


Figure 4.13: ELF file structure [153]

The dynamic object file created by the native compiler is parsed when the kernel is downloaded to the devices. The required functions are loaded into memory, based on the entries in the symbol table, and then the ELF parser checks that the file is of the correct binary format (for the micro-architecture in question) and raises an error if the file is incorrect for the target device. The ELF object file contains function sizes, which are stored in memory along with the functions themselves to allow the device dynamic loader to allocate the correct amount of memory in the heap. As the compiler has previously inserted the number of local variables and arguments into the function declaration code, the abstract machine is able to allocate the correct frame space for future function calls.

4.4.3 Summary

By leveraging the existing Olympus abstract machine memory model, the changes required to support dynamic loading within the generated C code are trivial. This is not to diminish the amount of work required to implement the ELF object file parser and the dynamic function symbol table / loader but serves to highlight the flexibility that the abstract machine and its associated memory model provide.

The dynamic loading model enables further memory savings, as well as supporting arbitrary sized codes, that in concert with the memory hierarchy work discussed

in Section 4.2, enables the deployment of complex Python codes to micro-core devices. A key benefit of the Olympus code generation and dynamic loading approach is that it provides the compiler with options to optimise the resulting code for run-time performance where static function dispatch is possible, using dynamic dispatch where required and also providing points for the loading (and unloading) of dynamic functions, based on the execution profile of the code.

Many aspects of the physical world may be modeled with Partial Differential Equations (PDEs) and lend a hand to predictive capability to aid the scientific discovery and engineering optimization. The HPCG benchmark is used to test an HPC machine's ability to solve these important scientific problems.

Dongarra et al., "A new metric for ranking high-performance computing systems" [155]

5

Experimental Environment

5.1 Overview

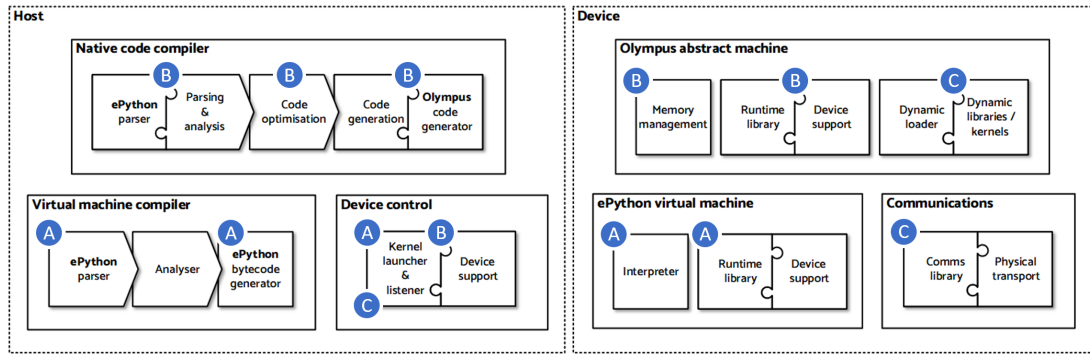


Figure 5.1: ePython architecture highlighting components evaluated by the benchmarks discussed in this chapter

This chapter describes the experimental environment used to evaluate the implementation of the enhancements to ePython to address RQ1 (How to manage the limited on-chip memory for data?), RQ2 (How to manage the limited on-chip memory for code?) and RQ3 (How to address the low runtime performance of virtual machines?), across a range of CPU architectures. Micro-cores introduce additional challenges over and above those for traditional CPUs, including the availability and access to these architectures for development and testing. Section 5.2 details the approach to selecting additional micro-core architectures in addition to the Adapteva Epiphany-III, including the design of a new FPGA-based micro-core design, Cerberus, discussed in Section 5.2.1. The design of Cerberus, a custom RISC-V multi-core CPU design based on the PicoRV32 and MicroBlaze soft-cores, also required the ability to quickly assess the applicability of existing soft-core designs, resulting in the creation of the Eithne micro-core benchmarking framework detailed in Section 5.2.2.

Figure 5.1 shows the ePython architecture, including the existing virtual machine and new Olympus native code generation abstract machine components and compiler, highlighting the components that were evaluated by the benchmarks discussed in Section 5.3.3. The support for arbitrary sized data (RQ1), including the communications model and memory hierarchy implementation, **A** in Figure 5.1, was evaluated by the cancer cell detection machine learning (ML) code discussed in Section 5.3.3.1. The techniques to increase runtime performance (RQ3) through enhancements to the VM and the introduction of native code generation (Olympus), including compiler structure, runtime memory model and type management **B**, were evaluated by the Fibonacci sequence, Jacobi iteration, LINPACK and Byte Sieve benchmarks in Sections

5.3.3.2, 5.3.3.3, 5.3.3.4 and 5.3.3.5, respectively. The dynamic function loading model  developed to support arbitrary kernel code sizes (RQ2) was evaluated by the Jacobi iteration code in Section 5.3.3.3.

5.2 Additional micro-core architectures

In order to address the portability challenge of Section 1.2 and confirm that the ePython architecture supports rapid porting to new micro-core architectures, additional platforms in addition to the Adapteva Parallella (Epiphany-III) were required. Three options were shortlisted for the project:

- The University of Washington’s RISC-V-based Celerity [156]
- The University of Manchester’s ARM-based SpiNNaker [157]
- A custom micro-core processor implemented on an FPGA

Each option was then evaluated in terms of capability, tool support and availability. The Celerity was rejected due to the extreme memory limitations, with only 4KB code and 4KB data on-chip RAM it was not possible to reduce the ePython virtual machine (VM) from 24KB to 4KB. The architecture was immature and there was a concern that undiscovered bugs may significantly impact project timelines. Furthermore, the project would have to rely on simulators and remote access to servers in the USA, with the associated risks that the simulator may differ in subtle ways from the actual hardware and the inability to access the platform could impact the project timelines. The SpiNNaker was rejected due to the specialised systems architecture, tooling and availability. For example, the SpiNNaker network message length is only 256 bytes, sending larger messages would require packaging, whereby the messages are split into packets on the source, tagged with sequence numbers and collated on the target. Furthermore, the networking design was optimised to support *spiking neural networks* and is not general purpose, resulting in failed network transfers if cores do not process messages within a short, preset time. The work required to manage message transfer failures, coupled with the lack of native support for messages over 256 bytes, was assessed as having a significant impact to the project timeline. Finally, the availability of the development board may be limited at crucial times due to its primary use for demonstrations and workshops. The FPGA micro-core implementation was selected due to the flexibility within the design, productionised tooling for the Xilinx PYNQ board, and local availability of the board that could meet the project timelines.

5.2.1 Cerberus soft micro-core processor

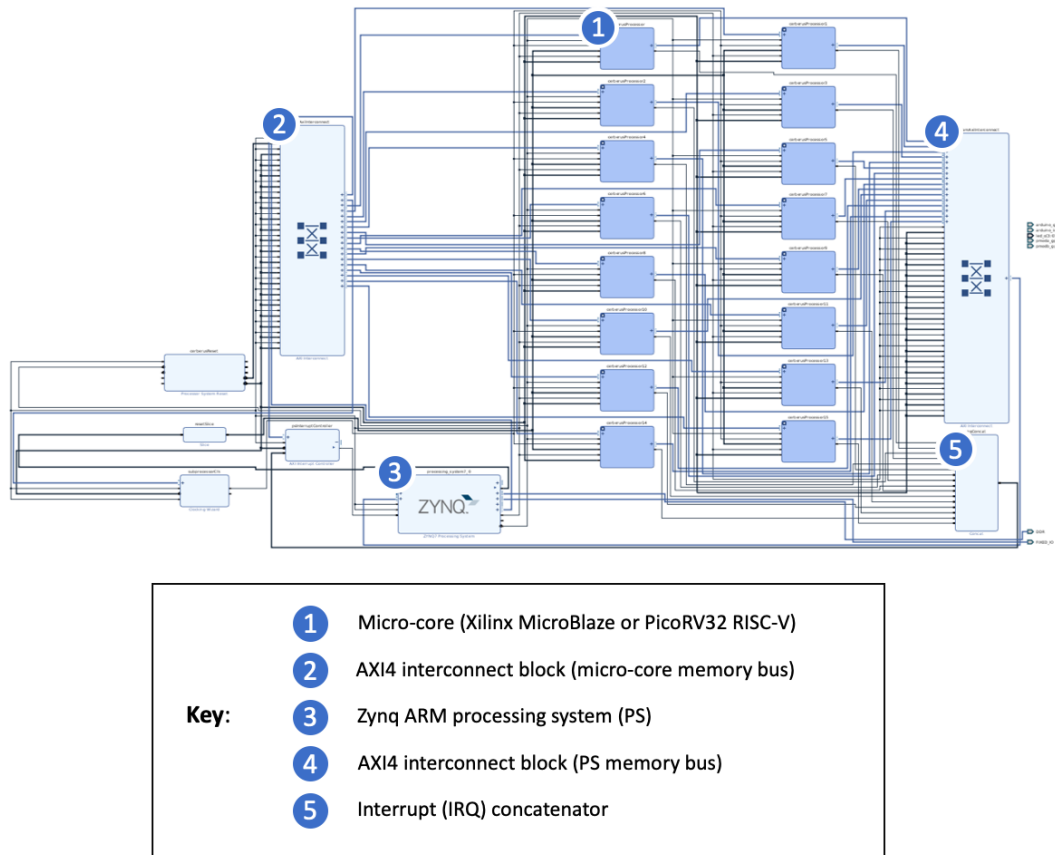


Figure 5.2: Cerberus 16-core RISC-V micro-core processor design

The decision to implement a micro-core architecture using a FPGA necessitated the design of a new soft-core accelerator with similar features to the Adapteva Epiphany micro-core, namely a number of inter-connected cores, each with a small amount (c. 64KB) of fast, on-chip scratchpad RAM and access to an area of slower, off-chip DRAM, shared with the CPU on the host. Although the micro-core accelerator design should be able to support a number of different soft-cores, the Xilinx MicroBlaze [158] and PicoRV32 [159] RISC-V soft-cores were chosen to support this research, with the latter being discussed in detail in this section. Whilst the MicroBlaze and PicoRV32 soft-cores are generally used as standalone CPUs in embedded designs, an inter-connected grouping of them¹, with associated limited scratch-pad RAM per core (32KB - 128KB), within a single IP block² visible to the host processor forms a

¹Each core can communicate with each other core through mapped memory areas within their local memory map, in a similar manner to the Epiphany-III micro-core.

²An IP (intellectual property) block is a reusable component within a FPGA design and can be *hard*

micro-core processor. The resultant design, Cerberus, is shown in Figure 5.2. Each individual micro-core, ① in Figure 5.2, is connected to a Xilinx Advanced eXtensible Interface v4 (AXI4) [160] *interconnect* IP block ② to form the micro-core processor bus that maps the individual core block RAM (BRAM), providing the on-chip scratch-pad memory, into the host CPU's memory map. A second AXI4 interconnect ④ forms the bus that connects the micro-cores back to the Zynq ARM Processing System ③, mapping the host shared memory area into each micro-core's memory map. The Xilinx Interrupt (IRQ) concatenator IP block ⑤ informs the ARM CPU that the micro-cores have faulted or completed processing. The other elements on the diagram are support components, including clock generators and interrupt controllers.

The detailed design of a single micro-core is shown in Figure 5.3. A PicoRV32 soft-core, ① in Figure 5.3, is connected to an internal AXI4 interconnect that enables connection to both the host shared memory and Port A of the local *dual-ported* BRAM ② via a Xilinx BRAM controller ③. The host CPU access to the local core RAM is provided via separate BRAM controller ④ connected to Port B of the BRAM. Although the PicoRV32 core is a von Neumann processor, where the instructions and data share a common bus, the Cerberus design also supports Harvard processors, such as the Xilinx Microblaze, where the instructions and data have separate buses, by using the AXI4 interconnect memory mapping to *alias* the instruction and data buses to the same BRAM block. PicoRV32 core access to the ARM Processing System (PS) DRAM is provided by the AXI Interconnect ⑤. Although the PicoRV32 can scale to a f_{max} of 450MHz on a 7-series Xilinx FPGA [159], the Zynq FPGA speed rating and Cerberus design limits the AXI and fabric clock ⑥ to 100MHz on the Xilinx PYNQ due to *negative slack* [161]. Negative slack is the condition where the timing constraints of a circuit are not met due to the distance between components and is often exacerbated by intermediate components in the path. It might be possible for the negative slack to be reduced enough to allow the f_{max} to be increased through the application of careful *floorplanning* [162], although 100MHz is sufficient for evaluating this research and, as such, was deemed out of scope.

The Cerberus design can be scaled from a single core up to 64 cores as a single accelerator. For this project, both RISC-V and MicroBlaze four, eight and 16 core Cerberus bitstreams running on the Xilinx PYNQ board were used for porting the ePython VM, the development of the native code generation runtime, benchmarking and testing. Cerberus is Open Source and the RISC-V versions are available for download (physical device) or *soft* (consisting of a collection of FPGA logic cells).

from GitLab [31].

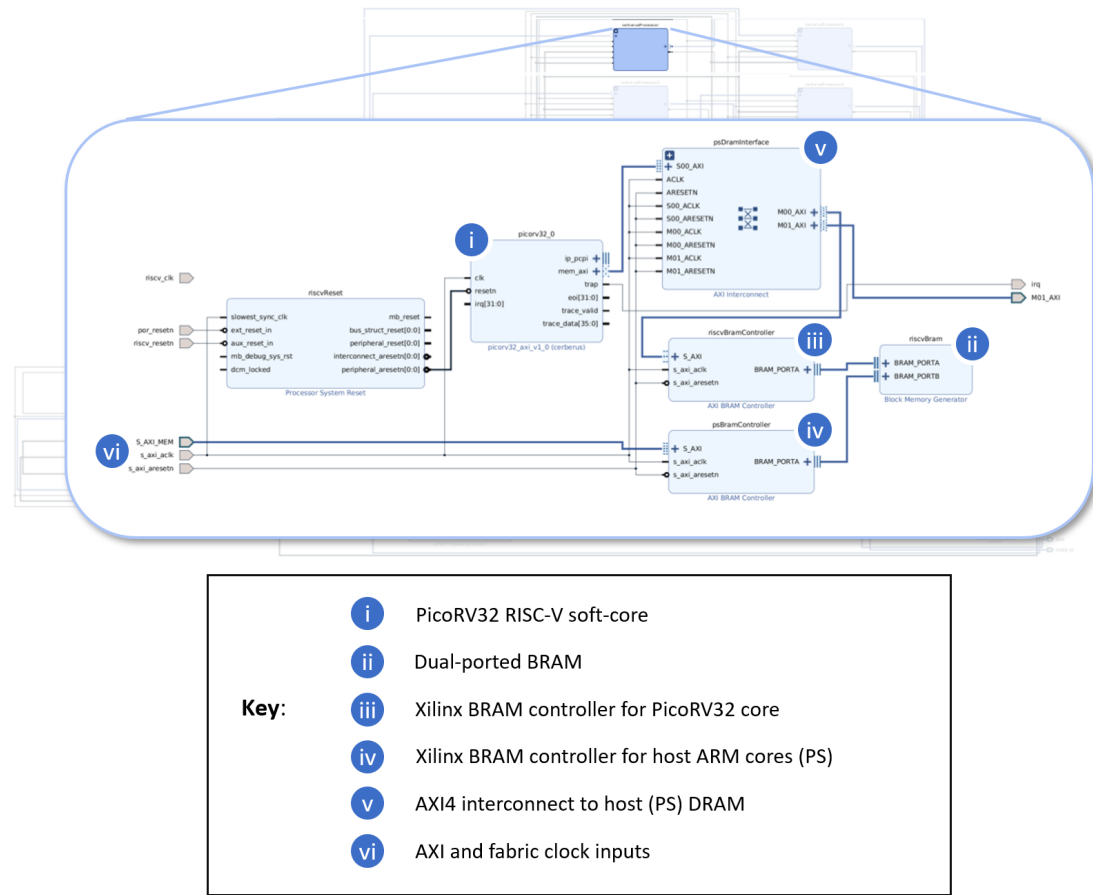


Figure 5.3: Cerberus RISC-V micro-core design

Figure 5.4 shows the PYNQ’s Zynq-7020 FPGA utilisation for the 1x32KB, 2x32KB, 4x32KB, 8x32KB, 12x32KB and 16x32KB RISC-V Cerberus designs. The green areas are the logic blocks for the Cerberus processor, the yellow areas are the clock generator blocks and the red areas are the AXI4 interconnect blocks. As expected, the Cerberus processor and BRAM blocks (green vertical bars) increase with the number of cores, with the AXI4 interconnect and clock generator blocks remaining relatively constant in size but being dispersed across the FPGA fabric. For the 16x32KB design, the limiting factor to the number of cores is not the general *Lookup Tables* (LUTs) on the FPGA, whose utilisation is around 67% but the on-chip BRAM utilisation at 91%. Therefore, as the BRAM allocation per core increases, the number of cores that can be implemented on the fabric decreases, irrespective of the amount of free LUTs. For example, the Zynq-7020 can only support a four core 128KB RISC-V Cerberus processor.

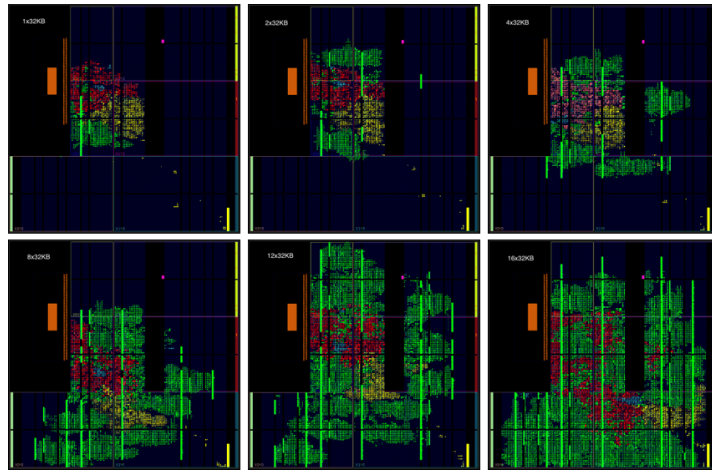


Figure 5.4: Cerberus RISC-V FPGA allocation

5.2.1.1 RISC-V soft-core selection

The selection of a soft-core for the Cerberus processor was not trivial as there were over 40 implementations of the RISC-V architecture alone at the time of design. Furthermore, the 32-bit RISC-V ISA has a number of extensions over the base integer ISA (RV32I), including:

- A - atomic instructions
- M - integer multiplication and division
- F - single-precision floating-point
- D - double-precision floating-point
- C - compressed instructions
- V - vector operations

These extensions (or variants) of the RISC-V ISA have runtime performance and power consumption trade-offs. They also have an impact in the amount of LUTs and timing constraints required to implement the processor on the FPGA fabric. Therefore, it is essential to be able to quickly assess the trade-offs of each option before committing to a particular ISA variation or a specific soft-core implementation of the same RISC-V ISA variation. Whilst it might appear that the simplest approach is to run a number of currently available benchmarks, the process is not as simple in practice. Even using a Cerberus design for a number of soft-cores, the tiny amounts of on-chip

memory make running traditional benchmarks difficult and the communications between the host CPU and micro-cores under test need to be addressed. Furthermore, many RISC-V soft-cores are not *verified* and may not implement all of the required features of the RISC-V ISA. For example, although the DarkRISCV is a pipelined RISC-V soft-core that is three times faster than the PicoRV32, it is not verified and does not fully implement the RV32I base ISA [163]. Partially-implemented ISAs can create issues when compiling kernels, as the compiler may use instructions from the ISA that are not supported by the unverified soft-core. These can be difficult to track down without disassembling the kernel binaries and comparing the generated instructions with those supported by the soft-core, which may not be documented and require interrogating its VHDL or Verilog source code. These issues make compiling a complex codebase, such as the ePython VM, hazardous on a new, unverified soft-core and a simpler approach is required.

As discussed in Section 5.2.1, the PicoRV32 soft-core supports the AXI4 bus, greatly simplifying integration with other IP blocks, and the host CPU, to create a micro-core accelerator. Based on the ease of integration and that it is verified for the RV32IMC ISA, which includes compressed instruction support that can reduce the size of compiled binaries by around 37% [164], the PicoRV32 was chosen to form the basis of the RISC-V Cerberus processor.

5.2.2 Eithne micro-core benchmarking framework

5.2.2.1 Architecture overview

To simplify the comparison of different micro-core architectures, a benchmark framework was required to quickly assess a micro-core’s suitability for the project. As a benchmarking framework for micro-core architectures did not exist, a new framework, Eithne³, was designed to meet this requirement. The deployment of codes to different micro-core architectures is time consuming due to the idiosyncratic nature of each micro-core architecture. For example, Adapteva provide *user space* libraries for loading and communicating with the Epiphany-III processor, whereas on the Xilinx PYNQ FPGA, kernel device drivers and the continuous memory area (CMA) library, which require superuser (`root`) access, are used for loading code and communicating with the soft-cores. By having a standard framework managing the code launch and communications, that is tested separately from the benchmark codes, a stable and consistent

³Eithne (/eɪnə/ "enya"): Gaelic for "kernel" or "grain".

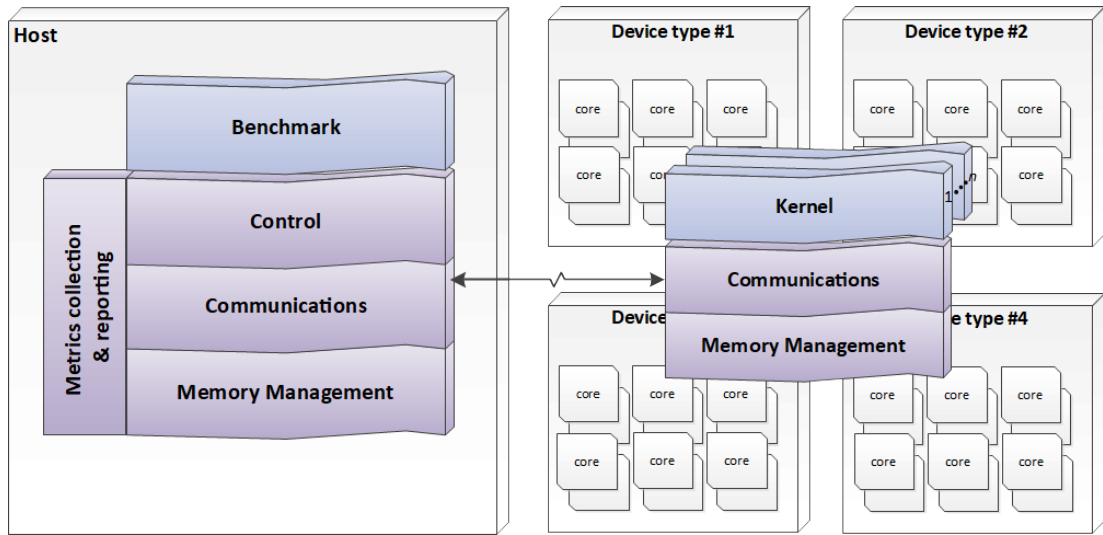


Figure 5.5: Eithne micro-core benchmarking framework architecture

environment for running codes can be provided, greatly simplifying the addition of new benchmarks. The architecture of the Eithne framework is shown in Figure 5.5. The framework has a layered *plug-in* architecture to support multiple benchmarks and devices, including simulating a device as a thread running on the host. The benchmark host and kernel codes are highlighted in blue, and sit on top of the purple components that encompass the Eithne framework and all the underlying hardware support. A set of application programming interface (API) calls provide kernel control, communications, memory management, as well as support for metrics collection and reporting. The layers can be reused across devices or a specific layer can be tailored for a device, as required, providing portability and configurability. Crucially, support for a new board or a new communications interface is just a new layer within the existing framework; different processors and architectures can be added without requiring changes to the other layers or the benchmarks themselves.

The control layer on the host manages the transfer and launch of kernels. The communications layer on both the host and device manage the transfer of data and control messaging. On the PYNQ and the Parallella boards, Eithne implements communications via shared memory between the host ARM processor and the micro-core device. Nascent support for communications over a serial link (UART) was built for the OpenISA RISC-V Vega board, with the communications protocol (Xfer) used for the shared memory implementation being extended to include software flow control (XON/XOFF) and a 32-bit cyclic redundancy check (CRC) of the packets transferred over the serial link.

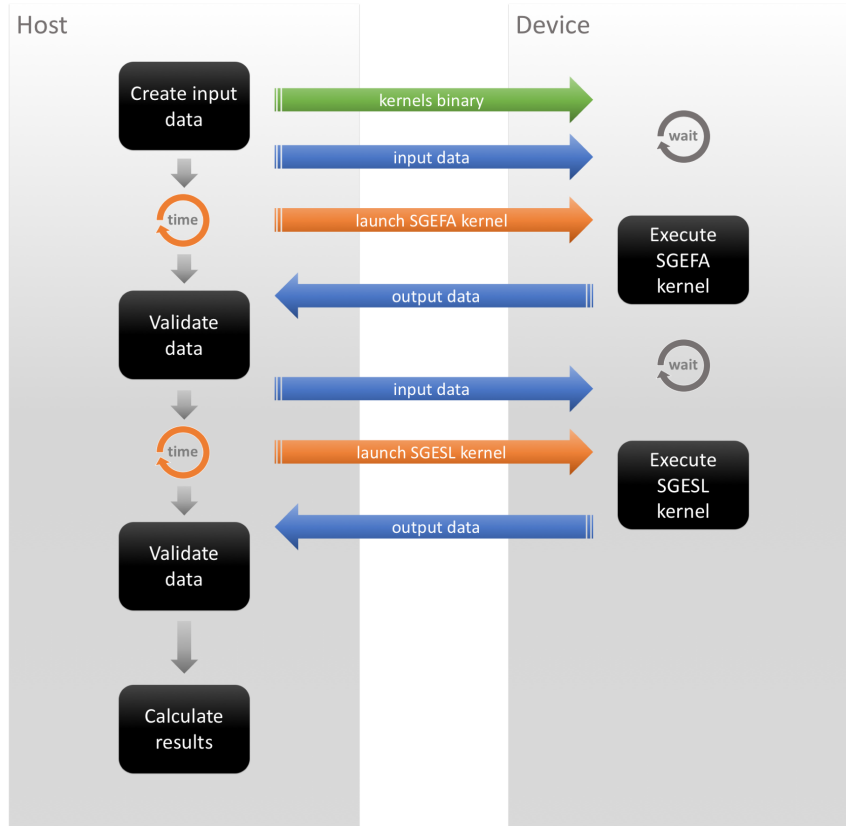


Figure 5.6: LINPACK flow using the Eithne framework

5.2.2.2 Host and device communications flow

A set of benchmarks (LINPACK, Jacobi, DFT and FFT)⁴ were updated to use the framework APIs by separating out the host driver code from the kernels. Once this was done, they were available on all devices that Eithne supports. Once modified to use the Eithne APIs, the benchmark driver code and kernels do not need to be modified to run on a new device, only relevant Eithne framework layers need to be updated to support that technology.

Figure 5.6 details the flow for the LINPACK benchmark using the Eithne framework. The input data is created on the host and the kernel binary is downloaded to the device and the communications listener is started⁵. Once the device listener has indicated to the host that it is ready, the input data for the first kernel is transferred to the device. Once this has been successfully transferred⁶, the kernel is launched on

⁴Available from the Eithne GitLab repository [25].

⁵When the Eithne framework is loaded on the host, the micro-cores are halted or placed in a wait state until an explicit API call is made to start them.

⁶The communications protocol always sends acknowledgment once a message has been actioned.

the device and its execution time is then recorded when complete. The kernel output data is transferred over to the host, where it is validated. Next, the input data for the second kernel is sent to the device and once transferred, the second kernel is launched and its execution time is recorded when complete. Finally, the second kernel's output data is transferred to the host and validated, and the benchmark results are calculated. Although the specific flow is different for each benchmark, the overall process is the same, with all the communications being initiated from the host, and that data transfer and kernel execution are separate allowing both to be timed individually.

5.2.2.3 Eithne API kernel modifications

```

1 void kernel_init(EithneTargetId id, EithneSharedMem buffer) {
2     EithneKernel kernels[] = { sgefa, sgesl };
3
4     EITHNE_INIT_DEVICE(vars, id, buffer + EITHNE_DATA_OFFSET, buffer,
5         kernels);
6
7     EITHNE_REGISTER_ARRAY(vars, A, EITHNE_FLOAT_ARRAY, a, N*LDA);
8     EITHNE_REGISTER_ARRAY(vars, B, EITHNE_FLOAT_ARRAY, b, N);
9     EITHNE_REGISTER_ARRAY(vars, IPVT, EITHNE_INTEGER_ARRAY, ipvt, N);
10    EITHNE_REGISTER_SCALAR(vars, JOB, EITHNE_INTEGER, job);
11    EITHNE_REGISTER_SCALAR(vars, RESULT, EITHNE_INTEGER, info);
12
13    EITHNE_START_LISTENER;
14 }
```

Listing 5.1: LINPACK kernel Eithne API calls

Listing 5.1 outlines the `kernel_init()` function added to the LINPACK kernel. This is called by Eithne when the kernel is loaded onto the device and sets up the communications buffer on the device, registers the variables that are to be shared with the host and starts the listener. The only other modifications required to the original kernel code *wrapping* of the `sgefa` and `sgesl` functions to pass the transferred variables as Eithne kernels have a void parameter list.

Details of the protocol are described in Section 4.2.3

5.2.2.4 Host driver code

```

1  buffer = EITHNE_ALLOC_MEM(sizeof(float)*N*LDA);
2
3  EITHNE_INIT_HOST(vars, HOST_ID, buffer + EITHNE_DATA_OFFSET,
4      buffer);
5
6  EITHNE_INIT_CORES(16);
7  EITHNE_START_CORES(16);
8
9  EITHNE_REGISTER_ARRAY(vars, A, EITHNE_FLOAT_ARRAY, a, N*LDA);
10 EITHNE_REGISTER_ARRAY(vars, B, EITHNE_FLOAT_ARRAY, b, N);
11 EITHNE_REGISTER_ARRAY(vars, IPVPT, EITHNE_INTEGER_ARRAY, ipvpt, N);
12 EITHNE_REGISTER_SCALAR(vars, JOB, EITHNE_INTEGER, job);
13 EITHNE_REGISTER_SCALAR(vars, RESULT, EITHNE_INTEGER, info);
14
15
16 EITHNE_SEND(vars, TARGET_ID, A);
17
18 t1 = cpu_time ( );
19 EITHNE_EXECUTE(TARGET_ID, SGEFA);
20 t2 = cpu_time ( );
21
22 EITHNE_RECV(vars, TARGET_ID, A);
23 EITHNE_RECV(vars, TARGET_ID, IPVPT);
24 EITHNE_RECV(vars, TARGET_ID, RESULT);

```

Listing 5.2: LINPACK host Eithne API calls

Listing 5.2 details the Eithne API calls that were added to the LINPACK host code. First, an area of shared memory large enough to transfer the biggest data item is created with the `EITHNE_ALLOC_MEM()` call, which also allocates an additional area for the communications protocol control area. Next, the Eithne framework is initialised, including the list of variables to be transferred, the ID for the host, the shared memory communications buffer and control area⁷. Then the micro-cores are initialised (the listener and kernels are downloaded) and then started using the `EITHNE_START_CORES()` call. Whilst the cores are waiting, we register the variables that we wish to transfer to and from the kernels using the `EITHNE_REGISTER` API calls. These APIs support a number of types, including arrays and complex numbers. The value of a variable,

⁷These can be in separate, disjoint areas of memory but are in consecutive areas of shared memory in most cases.

for example the input array A, is transferred using the `EITHNE_SEND()` and `EITHNE_RECV()` API calls, which take the target ID of a specific core on the micro-core device as an argument. The kernels are executed using the `EITHNE_EXECUTE()` call and the example shows how the timing of the kernel execution can be separated from the data transfer. If we want to time the communications to ascertain latency or bandwidth, we can wrap the appropriate messaging APIs within timing calls.

5.2.3 Eithne initialisation and data transfer timings

The Eithne API calls for the LINPACK host code in Listing 5.2 only time the `SGEFA` kernel execution and do not include the framework initialisation, variable registration and data transfer timings. However, it would be beneficial to understand the overhead of the Eithne framework initialisation and the data transfers between the host and device executing the kernels. Listing 5.3 illustrates the timing points for the Eithne initialisation and variable registration API calls on the host for the LINPACK benchmark. Listing 5.4 shows the timing points for the data transfers to and from the host to the device.

```

1  t1 = cpu_time();
2  buffer = EITHNE_ALLOC_MEM(sizeof(float)*N*LDA);
3
4  EITHNE_INIT_HOST(vars, HOST_ID, buffer + EITHNE_DATA_OFFSET,
    buffer);
5
6  EITHNE_INIT_CORES(16);
7  EITHNE_START_CORES(16);
8  t2 = cpu_time();
9
10 ...
11
12 t1 = cpu_time ();
13 EITHNE_REGISTER_ARRAY(vars, A, EITHNE_FLOAT_ARRAY, a, N*LDA);
14 EITHNE_REGISTER_ARRAY(vars, B, EITHNE_FLOAT_ARRAY, b, N);
15 EITHNE_REGISTER_ARRAY(vars, IPVT, EITHNE_INTEGER_ARRAY, ipvt, N);
16 EITHNE_REGISTER_SCALAR(vars, JOB, EITHNE_INTEGER, job);
17 EITHNE_REGISTER_SCALAR(vars, RESULT, EITHNE_INTEGER, info);
18 t2 = cpu_time ();

```

Listing 5.3: Eithne initialisation and registration code, with associated timing calls, for the LINPACK benchmark

```

1  t1 = cpu_time();
2  EITHNE_SEND(vars, TARGET_ID, A);
3  t2 = cpu_time();
4
5  ...
6
7  t1 = cpu_time();
8  EITHNE_RECV(vars, TARGET_ID, A);
9  EITHNE_RECV(vars, TARGET_ID, IPVT);
10 EITHNE_RECV(vars, TARGET_ID, RESULT);
11 t2 = cpu_time();

```

Listing 5.4: Eithne data transfer code, with associated timing calls, for the LINPACK benchmark

Table 5.1 details the timings for these points for the Olympus and native C LINPACK benchmark kernels, compiled at optimisation level `-O3`, running on the Adapteva Parallella. The Olympus kernel initialisation time on the host is, on average, approx-

imately 7% ($M = 1.4\text{ms}$) longer than for the native C kernel and can be explained by the additional time required to transfer the slightly larger Olympus kernel binary to the Epiphany-III device. The variable registration code on the host is identical for both the Olympus and native C kernels and although there is a 20% ($M = 1\mu\text{s}$) time difference between both kernel types, this difference can be attributed to other processes running on the host that impact the running of the kernel driver program.

Table 5.1: Mean (M) and maximum Eithne initialisation and data transfer times (seconds) for the LINPACK benchmark Olympus and native C kernels on the Adapteva Parallella (–03)

Eithne operation	Olympus mean ⁸	Olympus max	native C mean	native C max
Initialisation	0.01952	0.02647	0.01817	0.02522
Variable registration	0.000004	0.000019	0.000005	0.000019
Data transfer to device	0.001156	0.001198	0.001154	0.001198
Data transfer from device	0.000365	0.000406	0.000357	0.000405

The data transfers to and from the micro-core device are dependent on the listener thread on the host and are, therefore, impacted by other processes running on the host. The worst case difference between Olympus and native C kernels for the data transfer to the Epiphany-III is only 0.17% ($2\mu\text{s}$) and there is no measurable difference between the maximum transfer times. The maximum difference in transfer time between Olympus and native C kernels on the Parallella is only 2.19% ($8\mu\text{s}$) for the transfer from the device and the difference between the maximum transfer times from the device is only 0.25% ($1\mu\text{s}$).

One interesting aspect of the results detailed in Table 5.1 is the significant difference in data transfer times to and from the Epiphany-III, where data transfer to the device is around 3 times slower than the data transfer from the device. This is due to the host / device interface of the Parallella, where Epiphany core data *pulls* from the host DRAM are approximately 3 times slower than Epiphany core data *pushes* to the host DRAM, due to asymmetric bandwidth [107]. The Eithne framework utilises a listener running on the micro-cores which performs the host data transfers and consequently it exhibits this mismatch between data transfers to and from the host. However, on the Parallella this implementation is still advantageous as core-initiated data transfers exhibit up to a 27 times increase in bandwidth over host-initiated data transfers [107].

⁸Each kernel (Olympus and native C) was executed 100 times.

5.3 Test environment

5.3.1 CPU selection

In order to support the assessment of the implementation discussed in Chapter 4, a number of different platforms and processors were selected, including micro-cores and traditional CPUs, as detailed in Table 5.2. As processor ISAs can have a signif-

Table 5.2: Test environment board specifications, including CPU architecture, memory size and operating system

Machine	Arch	CPU	RAM	OS
Adapteva Parallella	32-bit	2x ⁹ 650MHz Cortex-A9 (ARM)	1GB	Ubuntu 16
Adapteva Parallella	32-bit	16x 600MHz Epiphany-III	32KB per core	bare metal
Xilinx PYNQ-Z2	32-bit	8x 100MHz MicroBlaze	64KB per core	bare metal
Xilinx PYNQ-Z2	32-bit	8x 100MHz PicoRV32 RISC-V	64KB per core	bare metal
Creator Ci20	32-bit	2x 1.2GHz Xburst MIPS32	1GB	Debian 10
HP 15-g093	64-bit	4x AMD A4-6210 AMD64	4GB	WSL ¹⁰
Sun ULTRA 5	64-bit ¹¹	1x 400MHz UltraSPARC-IIIi	512MB	Solaris 10
HiFive Unmatched	64-bit	4x 1GHz U740 RISC-V	16GB	Ubuntu 21

icant impact on both the compiled kernel runtime performance and binary size, the MIPS32 (Xburst), Intel x86 (AMD64), SPARC (UltraSPARC-IIIi) and 64-bit RISC-V (U740), were included as targets for comparison with the micro-core CPUs. The SPARC was selected as an alternative to the x86 as it has been used until recently for supercomputers in the TOP500 list [165] and the SPARCV8 LEON series of processors are used by the European Space Agency (ESA). Furthermore, it is a traditional Berkeley Reduced Instruction Set Computer (RISC) architecture with *register windows* that, for large C programs running on the SPARC, can show a 33% to 50% reduction in the number of load and store instructions generated over a non-register window RISC processor [166], such as the RISC-V, MicroBlaze, MIPS and PowerPC. The SPARCV9 UltraSPARC-IIIi was also chosen as a primary development target it is *big endian* and has strict byte alignment constraints that exercise the Merlin virtual machine and Olympus abstract machine implementations. Furthermore, this selection of CPUs tested portability between 32-bit and 64-bit processors with varying alignment

⁹Refers to the number of cores within the CPU.

¹⁰Windows Subsystem for Linux on Microsoft Windows 10

¹¹Olympus abstract machine and native C benchmark codes compiled to 32-bit binaries, with 32-bit integers and single-precision floating point on the SPARCV9.

constraints and byte ordering.

5.3.2 C compiler selection

The C compilers listed in Table 5.3 were used to compile and run the ePython virtual machine, the Merlin virtual machine, the Olympus abstract machine codes and native C benchmarks, to obtain the results for Chapter 6. Additional C compilers were used

Table 5.3: C compiler versions by CPU architecture

CPU	C compiler	Version
ARM Cortex-A9	gcc	5.4.0
Epiphany-III	e-gcc	5.4.0
MicroBlaze	mb-gcc	5.2.0
PicoRV32	riscv32-unknown-elf-gcc	8.2.0
MIPS32	gcc	8.3.0
AMD64 (x64)	gcc	9.4.0
SPARCv9	gcc	5.5.0
RISCV64	gcc	10.3.0

to test the portability of the Merlin virtual machine and Olympus abstract machine, including Clang [167] on macOS 10.15, the Small Device C Compiler (SDCC) [168] and OpenWatcom C [169] on Windows 10.

5.3.3 Benchmark selection

A number of benchmarks were selected to test the key areas of implementation: memory hierarchy support for arbitrary data, dynamic code loading for arbitrary-sized kernels and the native code abstract machine to increase kernel runtime performance. A description of each benchmark and the reason for their selection is provided in Sections 5.3.3.1, 5.3.3.2, 5.3.3.3 and 5.3.3.4. Table 5.4 details the mapping of benchmark execution to CPU.

For all benchmarks, the results reported in Chapter 6 were averaged over 100 execution runs.

5.3.3.1 Machine learning (ML) image processing

A parallel machine learning code for cancer cell detection within 3D scan images was chosen to test the runtime performance of the new pass by reference kernel data model,

Table 5.4: CPU / benchmark mapping

CPU	ML image	Fibonacci	Jacobi	LINPACK	Byte Sieve
ARM Cortex-A9	✓	✓	✓	✓	✓
Epiphany-III	✓	✓	✓	✓	✓
MicroBlaze	✓	✓	✓	✓	✓
PicoRV32	-	✓	✓	✓	✓
MIPS32	-	✓	✓	✓	✓
AMD64 (x64)	-	✓	✓	✓	✓
SPARCv9	-	✓	✓	✓	✓
RISCV64	-	✓	✓	✓	✓

discussed in Section 3.2.2.1. This code was also used to test the model’s viability in supporting real world applications on micro-cores using dynamic languages by providing support for arbitrary sized kernel data. The parallel ML code was run using the ePython VM using the 16 cores of the Epiphany-III and the 8 cores of the MicroBlaze micro-core architectures, and compared against equivalent single-threaded Python code running on the ARM Cortex-A9 and AMD64 processors.

5.3.3.2 Fibonacci sequence

A simple iterative Fibonacci sequence code [170] was used to determine the runtime performance and bytecode size of the different Merlin virtual machine opcode addressing modes, discussed in Section 4.3.2, and to compare the runtime performance of the Merlin VM against the Olympus abstract machine and native C. The Fibonacci code was written in Merlin assembly language, ePython and C, and run on a single ARM core. The Fibonacci code was also compiled to the Olympus abstract machine for comparison of the memory access and simple looping construct runtime performance between Olympus and native C on the CPUs listed in Table 5.4.

The listings for the Merlin assembler, ePython and C Fibonacci codes are in Appendix A and are available at [171].

5.3.3.3 Jacobi iteration

As [155] highlights, solving partial differential equations is common in engineering and scientific HPC codes. Therefore, a benchmark code for solving Laplace’s partial differential equation for diffusion via Jacobi iteration was chosen to compare the runtime performance of the ePython VM against the Olympus abstract machine code. It

was also used to compare the binary size and runtime performance of statically and dynamically loaded functions within Olympus kernels, at GCC compiler optimisation levels `-Os` and `-O3`. The serial Jacobi code was run on a single core, with a problem size (NX) of 100 and 10000 maximum iterations.

The ePython and C Jacobi iteration codes are available at [171].

5.3.3.4 LINPACK

The LINPACK benchmark [21] measures the floating point performance of a computer by solving a matrix problem using LU decomposition. It is a long-established benchmark, having been introduced in 1979, and is the standard benchmark used to rank supercomputer runtime performance for the Top500 [165] list. The C version [172] was selected to benchmark the runtime performance (MFLOPS¹²) of the Cerberus micro-core designs (MicroBlaze and PicoRV32 RISC-V) against the Epiphany-III. However, as LU factorisation is commonly used in scientific and industrial applications, such as design automation, machine learning and signal processing [173], LINPACK was also selected to compare the runtime performance of the Olympus abstract machine¹³ against native C, at GCC compiler optimisation levels `-Os` and `-O3`. For both the Cerberus and Olympus abstract machine investigations, a serial version of the LINPACK code was run on a single core. Due to the extremely small memory available on the micro-core devices, the problem size n was 50 and for traditional CPUs $n = 1000$. The code listings are in Appendix B.

The original LINPACK benchmark code can be found at [172] and the ePython and C versions are available at [171].

5.3.3.5 Sieve of Eratosthenes (Byte Sieve)

The Sieve of Eratosthenes benchmark [174], often referred to as *Byte Sieve*, is commonly used to test compiler code generation runtime performance and efficiency [175] [176]. The standard C and a new ePython version of the benchmark were used to determine the runtime performance of integer array access and looping constructs of the Olympus abstract machine relative to native C, to augment the Jacobi and LINPACK benchmarks that perform floating point array access and calculations. The standard Sieve benchmark is serial and both versions were run on a single core of all the CPUs

¹²Million Floating Point Operations per Second

¹³An ePython version of LINPACK was written and compiled to Olympus abstract machine code.

listed in Table 5.4. Like LINPACK, due to the limited memory on the micro-core devices, the flag array size was reduced ($SIZE = 4095$) on the Epiphany-III, MicroBlaze and PicoRV32 micro-cores. On the other CPUs, per the original benchmark, $SIZE = 8190$. Whilst the C version of the benchmark uses a byte (`char`) array to store the flags, the ePython version uses an (`int`) array, as this is the base numeric type. This results in the original flag array size requiring 32KB of memory due to the 32-bit integers on the micro-core devices. As this is the total memory available on the Epiphany-III for both data and code, the size of the flag array was reduced correspondingly.

The original Sieve benchmark code, in a number of languages including C, can be found in [174] and the ePython and C versions used in this work can be found at [171].

5.3.4 Benchmark rationale and approach

The ML image processing code discussed in Sections 5.3.3.1 and 6.2 was chosen to assess the practicability of running *real-world* codes on the ePython VM after the introduction, by this work, of the pass by reference kernel data model, host data prefetching, memory hierarchy and memkind additions discussed in Sections 3.2.2 and 4.2.

The key focus of the benchmarks discussed in Sections 5.3.3 and 6.3.2 was to determine the viability of the Olympus abstract machine on micro-core architectures in terms of binary code size and runtime performance. As C is the dominant alternative programming language for these devices, codes written in ePython and compiled to native code via the Olympus abstract machine were compared against native C versions in terms of binary code size and runtime performance. The Fibonacci sequence, Jacobi iteration, LINPACK and Byte Sieve benchmarks were run across a number of devices to determine any impacts to the binary code size or runtime performance of the Olympus abstract machine by the underlying CPU architecture. Furthermore, by running the benchmarks on a number of 32 and 64-bit RISC and CISC CPUs, including a mix of little and big-endian architectures, issues affecting the portability of the Olympus abstract machine could be determined and addressed.

The choice of benchmark was driven by the limited on-chip memory of the micro-core devices. For example, on the Epiphany-III, the LINPACK binary, compiled at optimisation level `-O3` is approximately 11KB, requiring the default Olympus heap size of 24KB to be reduced. Whilst this issue can be offset through the use of dynamic function loading, per the custom Jacobi benchmark discussed in Sections 6.4.1 and

6.4.2, the benchmarks were chosen for their binary codes to be compact enough to execute from a monolithic binary deployed to the micro-core devices.

The Fibonacci sequence benchmark was chosen as it is simple to code in assembly language, thereby enabling the comparison of the different addressing modes in the Merlin VM and a comparison of Merlin against the ePython VM, Olympus and native C. It was also used for Olympus abstract machine and native C binary code size and runtime performance comparisons. It was chosen as part of these investigations due to it being a simple iterative loop code that enables the comparison of basic integer operations and looping constructs as the static / dynamic function call dispatch overheads are not present, in contrast to more complex benchmark codes. The LINPACK and Jacobi iteration benchmarks solve Partial Differential Equations (PDEs) that are common in HPC codes [155] and are candidates for offloading to micro-core accelerators or GPUs. Furthermore, LINPACK is commonly used to measure the floating point performance of a machine, for example for the Top500 list [165]. In this work, LINPACK is used to determine the floating point operation overhead within the Olympus abstract machine in comparison to native C and to determine if the Olympus C source code elicits vector opcodes by the C compiler. The Jacobi iteration was chosen to test the floating point array access performance, as well as the runtime performance of native C math library function calls¹⁴ from within Olympus. The Sieve of Eratosthenes (Byte Sieve) was designed to test integer array operation, looping construct and control structure runtime performance [174].

Each Olympus and native C benchmark code were compiled at GCC optimisation levels `-Os` and `-O3` for each CPU architecture, resulting in 64 unique benchmark results. As Section 6.5 highlights, this selection of benchmarks was sufficient to evaluate Olympus in terms of runtime performance and binary code size, as well as determining the portability of the approach, across different CPU architectures.

¹⁴For the PicoRV32 Jacobi kernel, a custom C math library was created that contained only the code required to support the `sqrtf` and `powf` C functions.

The performance of a computer is a complicated issue, a function of many interrelated quantities. These quantities include the application, the algorithm, the size of the problem, the high-level language, the implementation, the human level of effort used to optimize the program, the compiler's ability to optimize, the age of the compiler, the operating system, the architecture of the computer and the hardware characteristics.

Dongarra et al., "The LINPACK Benchmark: past, present and future" [21]

6

Results

6.1 Overview

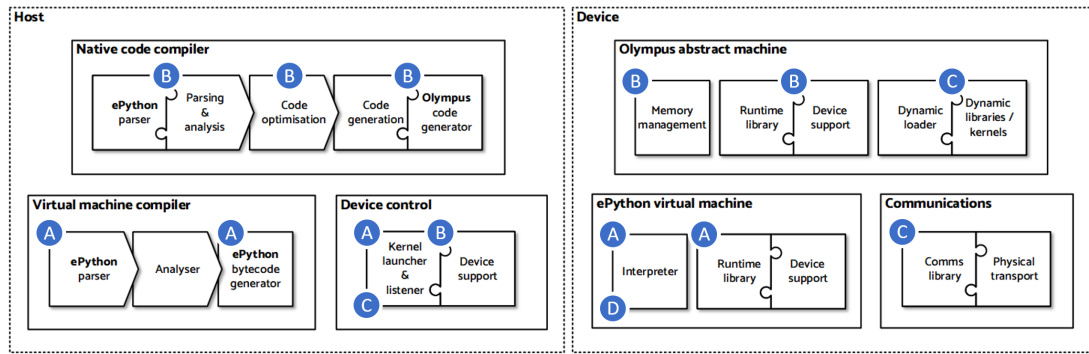


Figure 6.1: ePython architecture highlighting components evaluated by the benchmark results discussed in this chapter

This chapter presents the results of the benchmarks to assess the implementation of the enhancements to ePython to address RQ1 (How to manage the limited on-chip memory for data?), RQ2 (How to manage the limited on-chip memory for code?) and RQ3 (How to address the low runtime performance of virtual machines?).

Figure 6.1 shows the ePython architecture, including the existing virtual machine and new Olympus native code generation abstract machine components and compiler, highlighting the components that were evaluated using the benchmark results discussed in this chapter. Section 6.2 focuses on the work undertaken on memory hierarchies, **A** in Figure 6.1, to support arbitrarily large data (RQ1) within ePython kernels running on the VM. Section 6.3 discusses the size and runtime performance of Olympus native code kernels (RQ3) **B** against native **C** using a number of standard benchmarks across a number of processor architectures. Section 6.4 covers the runtime performance and usability of the dynamic function loading support (RQ2) within the Olympus abstract machine **C**, with particular focus on the Epiphany-III. The results of the investigations into the impact on runtime performance and bytecode size of the different addressing modes of the Merlin virtual machine (RQ3) **D** are discussed in Section 6.3.1.

6.2 Managing arbitrary data size (RQ1)

6.2.1 ML image processing

The pass by reference model provided by the memory hierarchy support within ePython enables the deployment of kernels that require arbitrary-sized data to be deployed to

the target micro-cores architectures. This was tested by running a Machine Learning application to detect cancer cells in 3D lung scan images from the 2017 Data Science Bowl [177] data science competition to develop lung cancer detection algorithms. The National Cancer Institute (NCI) provided thousands of high-resolution 3D lung scans to test the detection codes. In this work, the NCI images are used differently to the competition, where the evaluation of whether micro-core architectures and the parallelism that they provide can benefit the area of machine learning, rather than competition's concern regarding accuracy of prediction. A simple neural network with one hidden layer of 100 neurons which splits the 3D CT lung scans into two groups, 70% for training and 30% for testing was developed by [101]. Here, the input pixels of the image are distributed amongst the micro-cores for the acceleration of the linear algebra involved in training the model and the back-propagation. Each micro-core is operating on a separate part of the overall image in parallel and where previously each image was copied on to the micro-cores on kernel invocation, the new offload approach means that these images now remain in host memory with a reference to them passed to the micro-cores on kernel invocation. The previous eager copy approach was shown to perform competitively against Python and native implementations, but the limited memory of micro-cores meant that images had to be interpolated down from a maximum on-disk size of over 100MB to a size that the input data and neural network could fit within the shared memory area on the host. For example, 32MB on the Adapteva Parallella. Here, the same ML code is being used as a benchmark but the modified behaviour of kernel invocation as described in Section 3.2.2.1 means that we can run the full sized images for the first time.

The ML benchmark has two data transfer options: *on-demand* and *pre-fetching*. The former transfers the data from the host for each data request in the kernel running on the micro-cores. The latter option pre-fetches the data before it is required by the kernel code. Figure 6.2 illustrates runtime performance results for ePython with the new pass by reference model for both on-demand and pre-fetching, against the previous eager data copying on kernel invocation under ePython. Also included are runs on the ARM host using CPython for the kernels and a native implementation which calls into Numpy for the kernels, which has been compiled with GCC -O3 optimisation level. There is also an implementation via CPython on AMD64, where like the ARM, the AMD64 result is based on execution on a single core. As detailed in Table 6.1, for each configuration, there are two results; *feed forward* is the time taken to do a forward pass of the neural network and *combine gradients* is the time taken to calculate gradients

for a batch of training data. There is also the *model update* phase, which is the time taken to update the model with the gradients for the batch. However, these results are not discussed as they are not impacted by the memory management strategy (eager data copy, on-demand or pre-fetch) for the reason that the phase does not rely on data transfer from the host.

Table 6.1: Lung cancer machine learning feed forward and combine gradients phase mean (M) runtimes: small images (seconds)

Strategy / CPU	Feed forward	Combine gradients
Epiphany eager data copy	0.03728	0.04135
Epiphany on-demand	0.7678	0.7731
Epiphany pre-fetch	0.03110	0.02994
MicroBlaze eager data copy	0.02254	0.02773
MicroBlaze on-demand	0.2765	0.2796
MicroBlaze pre-fetch	0.02161	0.02125
ARM Python	3.846	3.795
ARM Native	0.01040	0.04748
AMD64 Python	0.2133	0.2098

The results in Figure 6.2, where the vertical axis is log scale, represent the mean (M^1) runtime values for 150 scaled down, interpolated, images processed using ePython on the Epiphany and MicroBlaze micro-cores, CPython on the ARM and AMD64, and native C on the ARM. For these experiments there are 3600 input pixels distributed amongst the micro-cores, with a hidden layer of 100 neurons. There are two key data structures, a matrix of input-hidden layer weights distributed among the micro-cores and a vector of hidden layer-output neuron weights. Each small image, passed for kernel invocations is 14.4KB. Forward feed involves a dot product on the weight matrix with the image, and a second dot product on the resulting values with the hidden layer-output neuron weight vector. Combining gradients, done for each image involves a dot product and an outer product. However, the model weights are not updated until after the batch of 10 images completes. For these small images, each kernel involves around 45000 floating point operations.

It can be seen that the original ePython kernel invocation version, *ePython eager data copy*, for both the Epiphany and MicroBlaze, compares favourably against the CPython and native versions, running on a single core, on the ARM host and AMD which is due to the parallelism provided by the Epiphany and MicroBlaze.

¹ M denotes the arithmetic mean and SD denotes standard deviation.

The *ePython on-demand* versions represent the benchmark relying on the modified behaviour described in Chapter 3, with accesses performed on-demand and not taking advantage of pre-fetching. The *ePython pre-fetch* results represent a version of the code using the new pass by reference model and pre-fetching optimisation.

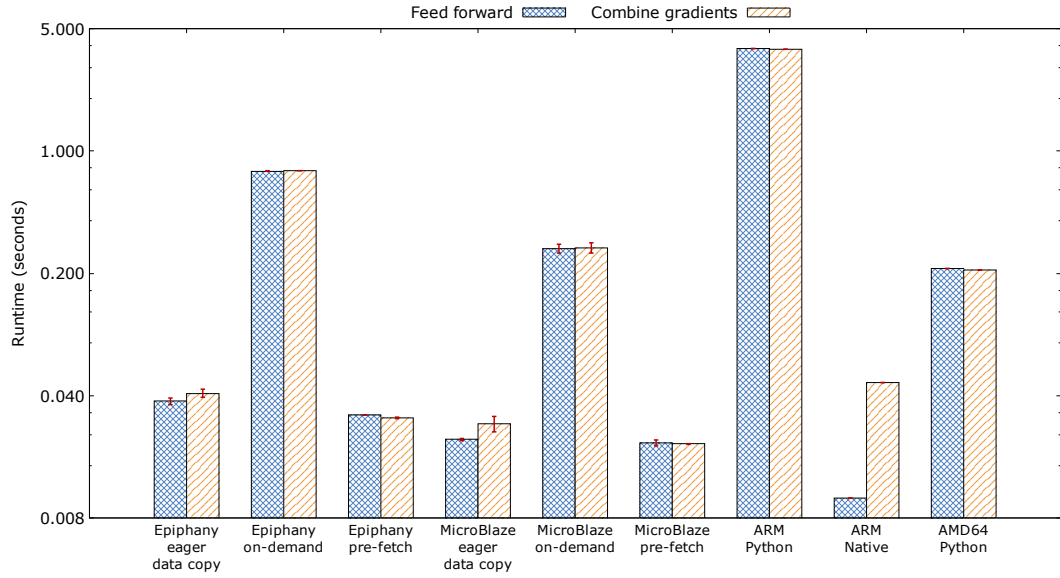


Figure 6.2: Comparing memory management strategies: Lung cancer machine learning runtimes: small images. Mean (M) runtime in seconds, with error bars denoting standard deviation (log scale)

For both the Epiphany and MicroBlaze, the on-demand version of this benchmark is significantly slower than the existing, eager data copy, behaviour of ePython. This is because the micro-cores retrieve individual elements of data, one at a time, and for each of these it must block until the transfer has completed. In contrast, the results in Table 6.1 highlight that the pre-fetch version of the benchmark provides on average up to 1.4 times ($M = 1.38$, $SD = 0.085$) better runtime performance for the calculation of gradients on the Epiphany than the existing eager data copy ePython implementation and is almost 25 times ($M = 24.69$, $SD = 0.002$) faster than the on-demand data copy approach for the Epiphany. The pattern is similar for the MicroBlaze, although the differences are less. The runtime performance improvement of pre-fetching over eager data copying is due to two factors, firstly the kernel can start as soon as the single reference is copied across rather than the entire data, and secondly the new data transfer mechanism enables the ePython module running in CPython to communicate directly with the ePython VM on the micro-cores rather than having to go via the ePython host process.

Table 6.2: Lung cancer machine learning feed forward and combine gradients phase mean (M) runtimes: large images (seconds)

Strategy / CPU	Feed forward	Combine gradients
Epiphany on-demand	1299	1308
Epiphany pre-fetch	40.10	40.92
MicroBlaze on-demand	549.4	553.7
MicroBlaze pre-fetch	44.49	41.95
ARM Python	2138	2301
AMD64 Python	352.0	328.0

Before the support for arbitrary-sized data, enabled by the introduction of memory hierarchy support in the ePython VM, it was impossible for these Python kernels to process the full sized images on the micro-cores. To test the impact of the on-demand and pre-fetch memory management strategies on the processing of large (full-sized) images by the same ML code used to process the small images, the code was run on a set of 20 different large images on the micro-core devices and compared with the serial Python code running under CPython on the ARM and AMD64 CPUs. Figure 6.3, where the vertical axis is log scale, illustrates the runtime performance of the machine learning benchmark for a forward pass through the neural network (feed forward) and calculation of gradients (combine gradients) in ePython on the Epiphany and MicroBlaze micro-cores, when running with the full sized images (again both on-demand and pre-fetch versions) and CPython on the ARM host and AMD64 using the same sized hidden layer as previously. Similarly to the small images, enabling the pre-fetching of data is much faster than the on-demand approach. From Table 6.2, the mean feed forward speed up is 13 times ($M = 12.60$, $SD = 0.004$) for the MicroBlaze and 32 times ($M = 32.33$, $SD = 0.136$) for the Epiphany. The results for combine gradients are similar, with a mean speed up of 13 times ($M = 13.20$, $SD = 0.005$) for the MicroBlaze and 32 times ($M = 31.96$, $SD = 0.109$) for the Epiphany. The full sized images are, on average, around 7 million pixels which is 1966 times larger than the small, interpolated, images of 3600 pixels. The average, single precision, input data that must be transferred to the micro-cores for each kernel is around 30 MB.

Crucially, not only does the support for memory hierarchies with data pre-fetch within the ePython VM allow the processing of full-sized images on the micro-cores but also, as Figure 6.3 illustrates, facilitates the runtime performance gains provided by running the ML code in parallel across a number larger number of cores, as found

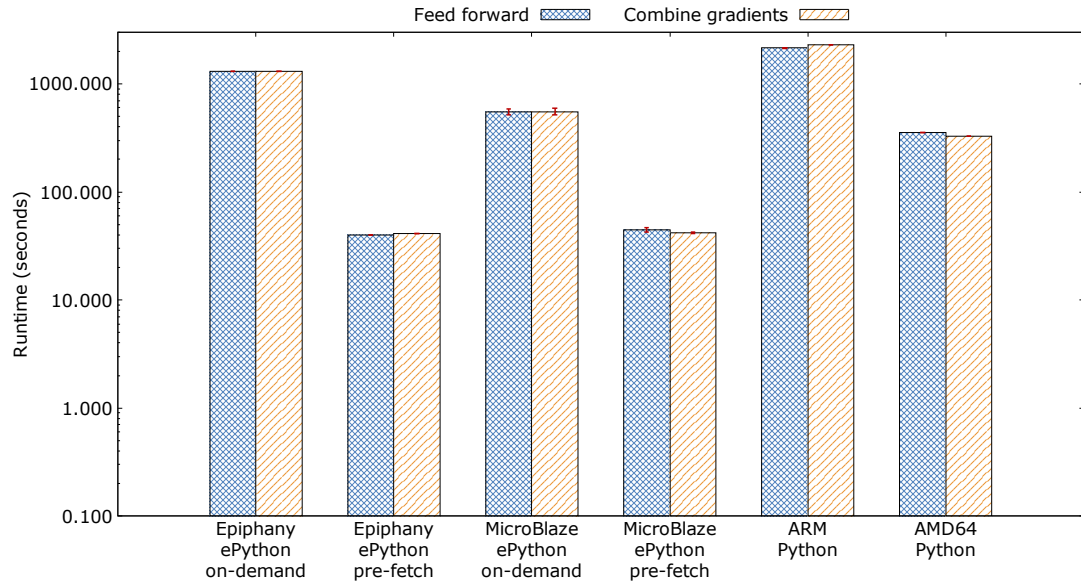


Figure 6.3: Comparing memory management strategies: Lung cancer machine learning runtimes: large images. Mean (M) runtime in seconds, with error bars denoting standard deviation (log scale)

on the Epiphany and MicroBlaze micro-cores.

6.3 Increasing kernel runtime performance (RQ3)

6.3.1 Merlin virtual machine

6.3.1.1 Fibonacci benchmark: kernel code size and runtime performance

Based on the runtime performance (Section 6.2) and memory footprint of the ePython VM (24KB on the Epiphany-III), it was hypothesised that a native code generation approach would be required to address RQ1 and RQ3, rather than enhancing the existing VM with a JIT compiler. In order to verify this hypothesis, which would determine the future development approach of the project, the Merlin VM (Section 4.3.2) was developed to determine if improvements could be made in terms of runtime performance and memory footprint. The Merlin VM was then compared with the existing ePython VM and native C using a single benchmark, a simple iterative Fibonacci sequence benchmark [170] (Section 5.3.3.2 and code listings in Appendix A) as this had to be hand-coded in low-level Merlin assembly language. This code was chosen due to it being a simple iterative loop code that enables the comparison of basic integer operations, conditionals and looping constructs; these are the fundamental operations

of a VM interpreter. More complex benchmark codes, such as LINPACK, introduce function call dispatch overheads that need to be avoided in order to be able to determine the *raw* runtime performance of the Merlin interpreter, specifically the dispatch loop (NEXT operation in Listing 4.2) and memory addressing / access. Fundamentally, the simple Fibonacci code assesses the baseline runtime performance and minimum binary code size / memory footprint of the interpreter.

It was also important to understand the impact of the different addressing models within Merlin. Therefore, stack-based, three-address and register-based opcode addressing versions of the Merlin Fibonacci benchmark were coded and compared in terms of runtime performance and bytecode size, as shown in Table 6.3².

Table 6.3: Merlin virtual machine / ePython VM comparison (ARM Linux -Os)

	ePython	Stack	Three-address	Register
Dispatches	N/A	20	12	12
Bytecode size (bytes)	131	85	86	66
Runtime size (bytes)	71576	6505	6505	6505
Execution time (secs)	253.6	25.31	11.35	10.37
Code saving (bytecode)	N/A	35.11%	34.35%	49.60%
Speedup	N/A	10.02x	22.35x	24.56x

Relative to the stack-based model, both the three-address and register-based models provide higher runtime performance, at 2.23 times and 2.44 times respectively. This increase in runtime performance is obtained by the 40% reduction in opcode dispatches for both models, due to the removal of stack operations. The sizes of the bytecodes for this benchmark are very similar (2% difference). The register-based bytecode is smaller than the stack-based and three-address bytecode due to the reduced number of memory load and store instructions required. For example, `ADDXY` adds the contents of the `X` and `Y` registers and stores them in the accumulator register (`A`), coupled with the use of the register-to-register copy instructions, Transfer `Y` to `X` (`TYX`) and Transfer Accumulator to `Y` (`TAY`) instructions, the memory load and store operations are removed. The Merlin assembly source for the stack-based, three-address and register-based Fibonacci code can be found in Appendix A, Listings A.1, A.2 and A.3, respectively. The source for the C, ePython and generated Olympus code are listed in Appendix A, Listings A.4, A.5, A.7, A.8 and A.9 respectively.

²All execution times are based on binaries produced by GGC C with the `-Os` optimisation level unless otherwise stated.

As shown in Table 6.3, Merlin provides a significant runtime performance improvement of between 10 to 25 times³ over the existing ePython VM, with the register-based code providing the highest runtime performance. This is combined with a bytecode size saving of between 35% to 50%, with, again, the register-based code returning the largest reduction with respect to ePython. These results indicate that significant runtime performance and bytecode size improvements could be made to the existing ePython VM by implementing the features of the Merlin register-based addressing model. In and of itself these results are interesting but as shown in Table 6.4, the fastest Merlin addressing model (register-based) is still between 11 to 12.5 times slower than native C. Furthermore, the overall code size for C is also up to 78% smaller than the combined bytecode and interpreter binary size for Merlin.

Table 6.4: Merlin VM, Olympus abstract machine and C binary comparison (ARM Linux)

	Merlin (-Os)	Merlin (-O3)	Olympus (-Os)	Olympus (-O3)	C (-Os)	C (-O3)
Bytecode size (bytes)	66	66	N/A	N/A	N/A	N/A
Runtime size (bytes)	6505	6744	4224	4252	1409	1491
Execution time (secs)	10.37	9.844	1.351	0.826	0.902	0.826
Code saving (total)	N/A	-3.64%	31.99%	36.95%	78.34%	77.89%
Speedup	N/A	1.05x	7.67x	12.55x	11.50x	12.55x

The Merlin virtual machine accounts for around one fifth of the 32KB of memory available on the Epiphany-III. Additionally, considering that this size does not include ePython runtime support or communications functions, the overall size of a JIT compiler based on a design similar to Merlin is unlikely to fit within the 32KB of memory available. Whilst this memory footprint could be ameliorated by running code from the off-chip shared memory (32MB per core), per the existing ePython VM, codes incur a significant runtime performance overhead when using this capability [19], in addition not all micro-core architectures provide a large shared memory space. Furthermore, the generation of the register-based code from Python source code is complex due optimal register allocation being *NP-complete*⁴ [178], making it a less attractive compilation target than the simple stack-based model, setting aside the runtime performance and size benefits of the former.

Table 6.4 highlights that the register-based addressing mode of Merlin is between

³The ePython script was pre-compiled to bytecode for separate execution, removing the parsing / compilation overhead.

⁴Nondeterministic Polynomial-time complete.

7 and 12.5 times slower than the Olympus native code abstract machine for the Fibonacci code, depending on the compiler optimisation level (`-Os` or `-O3`). However, the approximate 30% overall code saving provided by the Olympus abstract machine is, perhaps, a more interesting comparison as it includes functions to support dynamic languages that are absent from the native C code. The overall code size difference between Merlin and Olympus would be reduced for more complex kernels as the relative increase in bytecode size is likely to be smaller than that of the generated Olympus binary code [179]. Furthermore, to help free the precious on-chip memory further, a dynamic loader for Merlin could also be implemented to allow both virtual machine functions and kernel bytecode to be loaded, and unloaded, as required.

The Fibonacci code provides the baseline runtime performance and binary code size results for the interpreter. Whilst this is a simplistic benchmark, it clearly demonstrates that native code generation provides kernels that are up to 37% smaller and up to 12.5 faster than the fastest memory addressing (register-based) model of the Merlin VM, which more complex benchmarks would obscure due to additional overheads involved in execution. This benchmark alone is sufficient to conclude that a native code generation approach can provide higher runtime performance at a lower memory footprint than a new VM with a JIT because additional benchmarks, for instance alternative simple looping and memory access codes, would provide limited additional insights due to the same underlying VM operations and memory access pattern.

6.3.2 Olympus abstract machine

It was necessary to validate the Olympus abstract machine by evaluating the runtime performance and memory footprint of the generated code for standard scientific kernels on the micro-core devices, rather than just on the host. This was achieved by reusing the Eithne micro-core benchmarking framework, discussed in Section 5.2.2, originally developed to benchmark micro-core architectures and configurations. The framework runs on a number of hard and soft processor architectures, with a smaller and less complex codebase than the ePython runtime, coupled with a simpler communications model. Therefore, porting to a new processor or architecture is easier and quicker, requiring much less effort than porting the complete ePython runtime, enabling kernels to be deployed quickly for prototyping and testing. This was vital to ensure the Olympus abstract machine was able to support the target RISC-V (PicoRV32) micro-cores that were unsupported by ePython the time this research commenced. As Eithne de-

ploys and executes C-based kernels, Olympus test codes were wrapped by Eithne API calls for testing on the supported architectures.

6.3.2.1 Fibonacci benchmark: kernel code size

Table 6.5: Fibonacci kernel binary code size (bytes)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	3682	4678	1942	2886
MicroBlaze	3912	4976	3392	4476
PicoRV32	19006	19450	900	1328
ARM32	10306	10966	5284	5903
MIPS32	16761	17458	8438	9227
AMD64	16528	17904	7914	9274
SPARCv9	16126	16930	8478	9310
RISCV64	12411	13195	6608	7329

The simple Fibonacci sequence code (Section 5.3.3.2), used to compare the Merlin addressing models in Section 6.3.1, was repurposed to compare the memory access and looping runtime performance, along with the kernel binary code size, between Olympus and native C kernels at compiler optimisation levels `-Os` and `-O3`. The kernels were run on the Adapteva Epiphany-III, Xilinx MicroBlaze and PicoRV32 RISC-V micro-cores, and the ARM Cortex-A9, MIPS32, AMD64, SPARCv9 and 64-bit RISC-V traditional CPUs. The Olympus and native C kernel binary sizes for the Fibonacci code are detailed in Table 6.5. On the Epiphany-III, the Olympus kernel binary code size ranges from 1.6 times bigger than the native C kernel at `-O3` to around two times bigger at `-Os`. The MicroBlaze shows a similar pattern, with the Olympus kernels being 1.1 times bigger than native C at `-O3` and 1.15 times bigger at `-Os`. As Figure 6.4 shows, the kernel binary size difference between Olympus and native C at the different compiler optimisation levels is more consistent, at around two times bigger, for the traditional CPUs. For example, the Olympus kernels are between 1.8 (`-O3`) and 1.87 (`-Os`) times bigger than native C on the RISCV64. Likewise, on the SPARCv9, the difference ranges between 1.82 (`-O3`) and 1.90 (`-Os`) times bigger than native C, and on the MIPS32 the range is between 1.89 (`-O3`) and 1.99 (`-Os`) times bigger. The striking difference between the Olympus and native C kernel binary code sizes on the PicoRV32 is due to the peculiarity of the custom GNU *linker file* (Listing E.1 in Appendix E) required by the PicoRV32 and its inclusion of space for the Olympus heap

(16384 bytes). The impact of the custom PicoRV32 linker file on the reported Olympus kernel binary size is explained in detail in Section 6.3.2.7 (Byte Sieve benchmark).

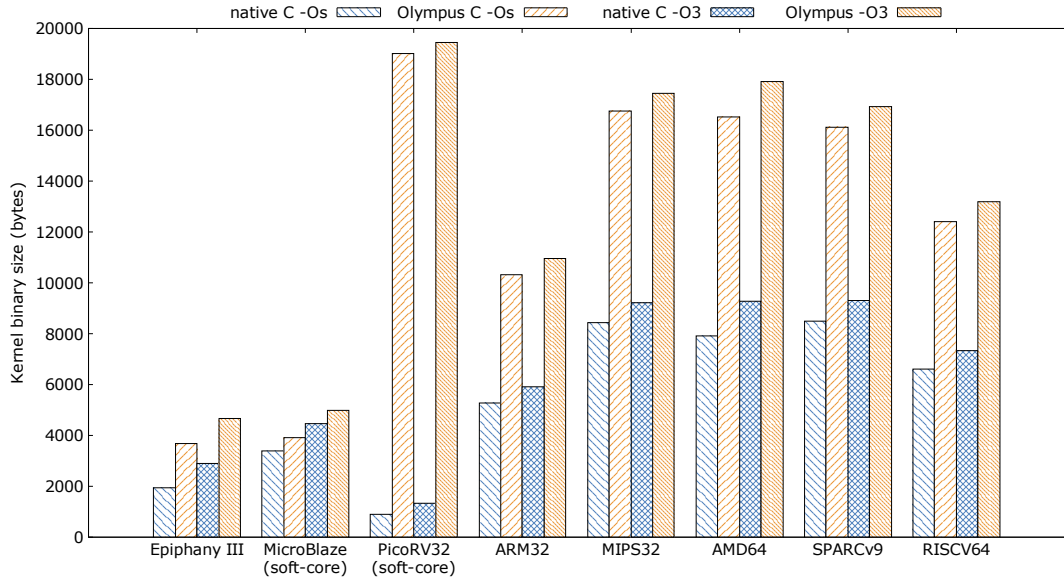


Figure 6.4: Comparing native C and Olympus kernel code size across architectures: Fibonacci benchmark

Figure 6.4 highlights that the Olympus kernels compiled at `-Os` are not significantly smaller than those compiled at `-O3`, with the average (M) difference in size between Olympus and native C kernels compiled at `-Os` and `-O3` of 11.76%. Considering the binary code size of the Epiphany-III Olympus kernel compiled at `-O3` is 4678 bytes, coupled with the default 16KB heap available, the Olympus native code abstract machine looks to provide a viable alternative, for simple codes, to the ePython VM, where the VM size of 24KB on the Epiphany-III restricts the available heap to 3840 bytes (Figure 2.10). Although the Olympus kernel binaries are, at a worst case, on average ($M = 183.6\%$) approximately two times larger than the native C counterparts, the small absolute binary size of simple Olympus kernels⁵ in comparison to the memory footprint of the ePython VM, and thereby enabling significantly more available memory for kernel data, is the key takeaway of these results.

6.3.2.2 Fibonacci benchmark: runtime performance

From Table 6.6 it can be calculated that, on the Epiphany-III, the native C kernels runtimes are between parity (1.0) and 1.6 times faster than Olympus kernels, at com-

⁵Excepting the PicoRV32 for the reasons detailed in Section 6.3.2.7.

Table 6.6: Fibonacci kernel mean (M) runtime (seconds)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	1.333	0.7498	0.8329	0.7499
MicroBlaze	13.50	3.499	4.499	3.499
PicoRV32	95.98	29.99	49.99	36.49
ARM32	1.239	0.4507	0.5255	0.4506
MIPS32	1.501	0.4170	0.6677	0.4174
AMD64	0.3407	0.1607	0.1931	0.1615
SPARCv9	4.349	0.8630	0.6163	0.8586
RISCV64	0.9009	0.3993	0.4992	0.2982

piler optimisation levels $-O3$ and $-Os$, respectively. A similar range is demonstrated on the AMD64, where the native C Fibonacci kernels are between parity (1.0) and 1.8 times faster than the Olympus kernels, at $-O3$ and $-Os$, respectively. The largest range is found on the SPARCv9 where the native C kernels are between parity (1.0) and 7.1 times faster than Olympus. The latter figure for the SPARCv9 is interesting as it suggests that the GCC compiler optimisations performed at $-Os$, that exclude the optimisations that impact the binary code size [180], are unable to optimise the Olympus abstract machine code effectively, in comparison to the excluded optimisations that are available at $-O3$. The error bars on Figure 6.5 show that although the average (M) kernel runtimes for Olympus and native C kernels for the SPARCv9 and AMD64 at $-O3$ are almost identical, there is variance in the runtime figures, $SD = 0.060$ for Olympus and $SD = 0.055$ for native C on the SPARCv9 and $SD = 0.012$ for both Olympus and native C on the AMD64. Similarly, on the ARM32 and MIPS32, the Olympus average (M) kernel runtimes are parity with native C ($M = 1.000$ on the ARM32 and $M = 0.999$ on the MIPS32) but the variance is significantly smaller at $SD = 0.00029$ for Olympus and $SD = 0.00020$ for native C on the ARM32 and $SD = 0.00140$ for Olympus and $SD = 0.00050$ for native C on the MIPS32. The $-Os$ and $-O3$ variance for the average (M) runtime figures for the ARM32 and MIPS32 are of the same scale as those for the Epiphany-III, MicroBlaze and PicoRV32. This is surprising considering that the ARM32 and MIPS32 CPU kernels are running on multitasking operating systems, per the SPARCv9 and AMD64, whereas the micro-core (Epiphany-III, MicroBlaze and PicoRV32) kernels are running on *bare metal*, where very little variance would be expected on such devices as no other codes are executing on the micro-cores at the same time as the kernels.

The $-O3$ PicoRV32 (RISCV32) kernel runtime results are surprising, where Olym-

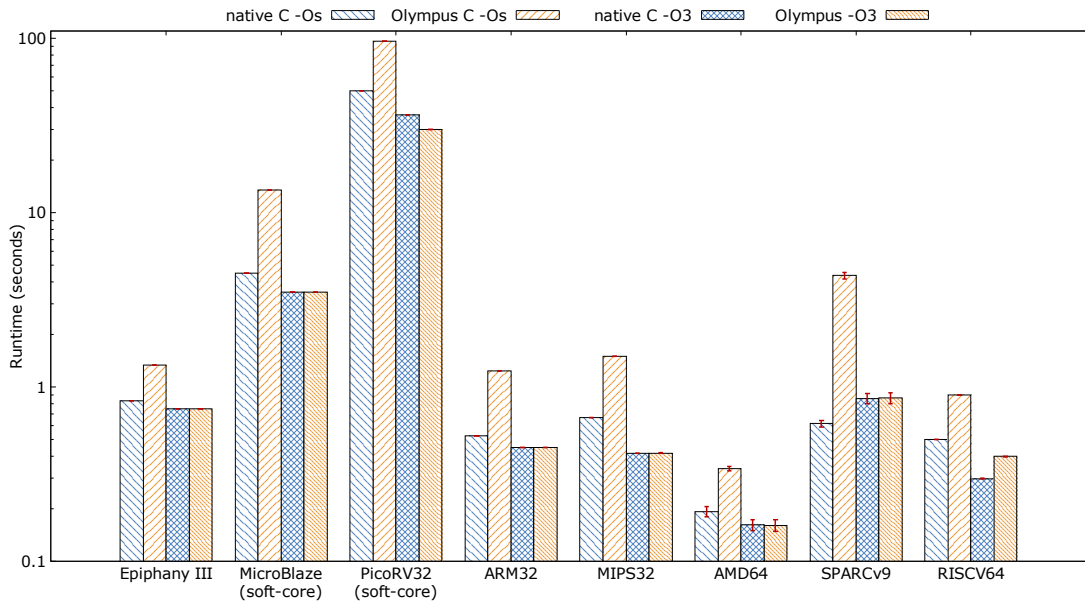


Figure 6.5: Comparing native C and Olympus runtime performance across architectures: Fibonacci benchmark. Mean runtime (M) in seconds, with error bars denoting standard deviation (log scale)

pus is 1.22 times faster than native C, especially when the reverse is found on the RISCV64, where native C is 1.34 times faster than Olympus. The PicoRV32 result suggests that the 32-bit RISC-V compiler optimisations at `-O3` for the native C kernel might be impacted by the generation of the compressed (RV32C) instructions, favouring the Olympus code. The RISCV64 is the only CPU where the Olympus Fibonacci kernel runtime cannot match native C, suggesting that the GCC compiler is unable to optimise the Olympus abstract machine code effectively for the 64-bit RISC-V ISA. Detailed investigation of the `-O3` Olympus and native C kernel disassembly listings across all the CPUs would be required to ascertain if the relatively poor optimisation of the Olympus code at `-O3` on the RISCV64 is due to a specific characteristic of the Olympus abstract machine code or the 64-bit RISC-V architecture.

Overall, the key takeaway from the simple Fibonacci sequence Olympus kernel runtime performance results is that the Olympus abstract machine integer memory access model and looping constructs do not impose a significant runtime overhead relative to native C, across the majority of CPUs used to execute the benchmark.

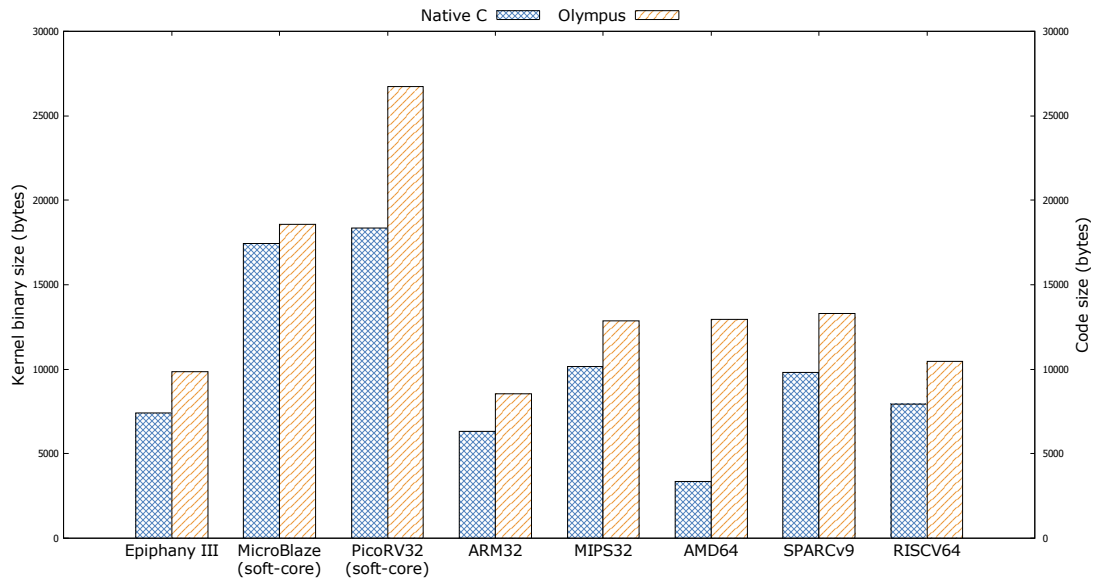


Figure 6.6: Comparing native C and Olympus kernel code size (-O3) across architectures: Jacobi benchmark

6.3.2.3 Jacobi benchmark: kernel code size

As shown in Figure 6.6, the speed optimised Olympus binaries are similar in size to native C on the Epiphany-III, MicroBlaze, MIPS32 and SPARCv9, with the Olympus binary sizes between 6-31% larger than the native C binaries on the micro-core architectures. However, there is a marked difference on the the AMD64, where the Olympus binaries are 74% larger. The AMD64 results in comparison to the other 64-bit CPU (SPARCv9) suggest that the C compiler for the x86 is more capable at optimising the native C kernel for size as well as runtime performance. The MicroBlaze Olympus kernel binary is only about 6% larger than the native C kernel binary, as highlighted by the results in Table 6.7. In this case, the Olympus abstract machine overhead may be small in comparison to the kernel function codes. Considering that the kernel binaries were compiled for runtime performance (-O3) rather than size (-Os), the overall Olympus kernel binary size is viable on the micro-core architectures as they all require less than 50% of the available on-chip RAM and are less than a third bigger than the equivalent native C kernel binaries.

These results significantly enhance the usability of the Olympus approach; the resulting binaries have, on average, less than a 42% runtime performance overhead over native C but still require less than 50% of the extremely limited memory of the target micro-core devices, providing significantly more available memory for kernel data

Table 6.7: Jacobi kernel binary code size (bytes -O3)

CPU	Olympus	native C
Epiphany-III	9870	7398
MicroBlaze	18588	17428
PicoRV32	26728	18344
ARM32	8537	6329
MIPS32	12844	10160
AMD64	12932	3350
SPARCv9	13287	9798
RISCV64	10477	7952

than the previous VM approach. Furthermore, as will be discussed in Section 6.4.1, there are options to reduce the initial Olympus kernel memory footprint further through dynamic code loading.

6.3.2.4 Jacobi benchmark: kernel runtime performance

A standard Jacobi code from [181], that is bundled with the Eithne framework [29], was written for Python and the runtime performance of the native C and Olympus versions were compared across a number of CPUs, as shown in Figure 6.7. As the

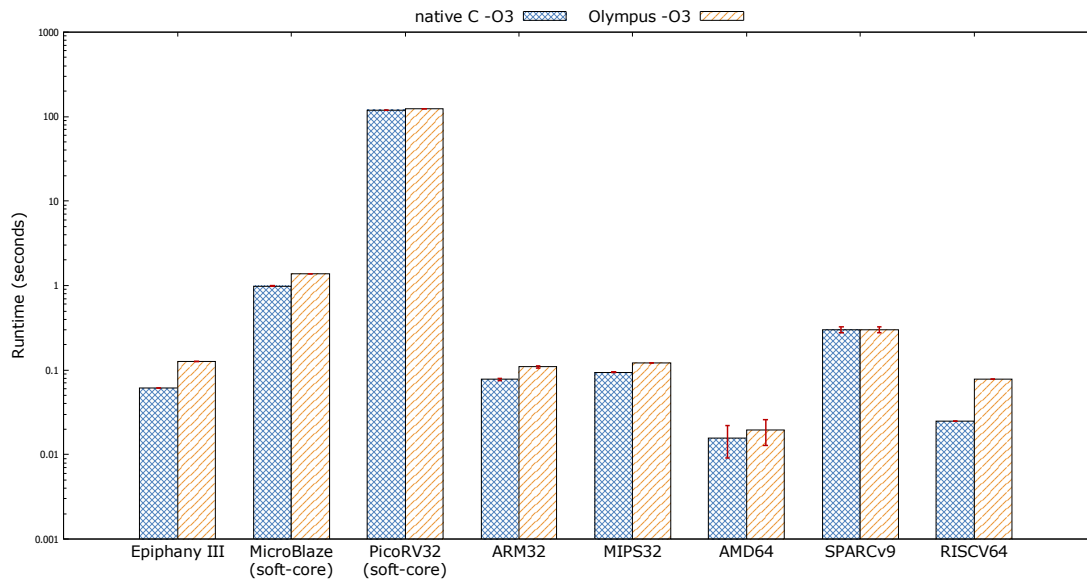


Figure 6.7: Comparing native C and Olympus runtime performance (-O3) across architectures: Jacobi benchmark. Mean runtime (M) in seconds, with error bars denoting standard deviation (log scale)

main concern was to assess the code generation runtime performance improvements, a single-core, serial version of the benchmark, with a problem size (NX) of 100 and 10000 maximum iterations was run on all platforms⁶. As expected, the native C kernels were faster than the Olympus versions, as detailed in Table 6.8. Across the eight CPUs, the native C kernels were 1.6 times quicker on average ($M = 1.57$), with the Olympus kernels on the PicoRV32 and SPARCv9 matching the runtime performance of the native C kernels. This is due to the small problem size and simple codes, reflecting the results for the Olympus and native C Fibonacci codes listed in Table 6.6. When the problem size (NX) was increased to 500, the SPARCv9 Olympus runtime was 14% slower than the native C version. Comparing this result to the average 43% runtime performance overhead of the Olympus abstract machine over native C on the other processors suggests the Olympus environment model may be leveraging the advantage of the SPARCv9's register windows [166].

Table 6.8: Jacobi kernel mean (M) runtime performance (seconds -O3)

CPU	Olympus	native C
Epiphany-III	0.1255	0.06107
MicroBlaze	1.377	0.9904
PicoRV32	122.8	119.8
ARM32	0.1093	0.07733
MIPS32	0.1218	0.09433
AMD64	0.03589	0.01563
SPARCv9	0.3006	0.3019
RISCV64	0.07781	0.02479

The runtime performance of the Epiphany-III relative to the MIPS32 and SPARCv9 processors is surprising, especially when the clock rate of the XBurst MIPS32 core (1.2GHz) is twice that of the Epiphany-III (600MHz). However, this is partly explained by the fact that the kernels are executing *bare metal* on the Epiphany-III, MicroBlaze, and PicoRV32, whereas they are running as POSIX threads on the MIPS32 (Linux), AMD64 (Windows Subsystem for Linux) and SPARCv9 (Solaris). Furthermore, the Epiphany-III is a superscalar design that can execute two floating point operations and one integer instruction per clock cycle [13], in comparison to the more simple pipelined architecture of the XBurst [182].

⁶The problem size of 100 is significantly smaller than the default size of 1000 but was required to enable the data to fit within the limited on-chip memory of the micro-cores.

6.3.2.5 LINPACK benchmark: kernel code size

As mentioned in Section 5.3.3, LINPACK was selected to exercise the Olympus abstract machine, over and above the initial investigations using the Jacobi benchmark in discussed Sections 6.3.2.4 and 6.3.2.3. The key issue is to determine if specific memory access patterns impact the size or runtime performance of the code generated for the abstract machine. For example, whether significant array access operations and calculations stimulate the C compiler to generate vector instructions, such as the Vector Multiply (VMUL), Vector Multiply Accumulate (VMLA) and Vector Multiply Subtract (VMLS) from the ARM NEON SIMD instruction set extension [183] or does the Olympus abstract machine mnemonics prohibit their use by the compiler, thereby impacting runtime performance?

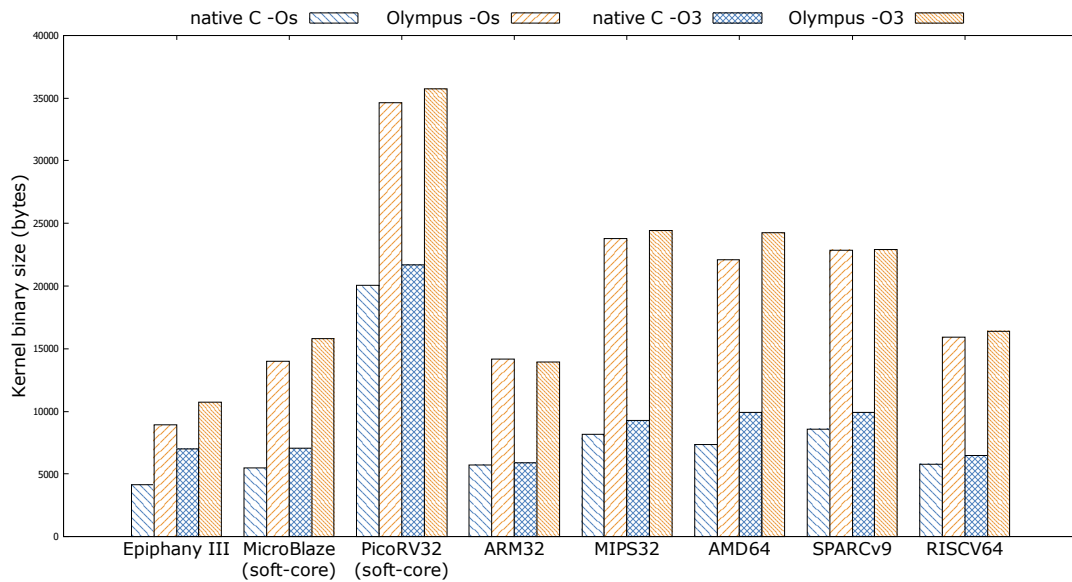


Figure 6.8: Comparing native C and Olympus kernel code size across architectures: LINPACK benchmark

The same approach as with the Jacobi benchmark was used, where the source ePython codes were compiled to C source and wrapped by Eithne API calls for execution on the target architectures. Figure 6.8 shows the size of the Olympus kernels relative to the hand-coded native C versions when compiled at optimisation levels `-Os` and `-O3`. The diagram illustrates that the C kernels are significantly smaller than the Olympus kernels on all platforms, at both compiler optimisation levels, for the LINPACK benchmark. Table 6.9 highlights that the difference in kernel size ranges from around 1.5 times bigger than native C on the Epiphany-III to 2.6 times bigger

on the MIPS32, using `-O3`. Interestingly, the difference ranges from around 2 times bigger than native C on the Epiphany-III to around 3 times bigger on the MIPS32 and AMD64. As these results are similar to those for the Jacobi kernels, this suggests that the Olympus mnemonics generate *wordy* C code, whereby a significantly larger number of underlying operations (machine opcodes) are generated by the C compiler in comparison to the equivalent native C operation. Furthermore, this wordy code is harder for the C compiler to optimise for size, resulting in the elevated range in comparison to optimisation level `-O3`. However, it should be noted that the Olympus kernels include a full compacting heap manager and other runtime functions required to support the dynamic features of ePython that are absent from the static native C LINPACK kernel.

The LINPACK results need to be set in context; although the best case `-O3` Olympus kernel size on the Epiphany-III micro-core is 50% bigger than the native C kernel, this only amounts to a 3.7KB overhead. The 10KB Olympus kernel binary allows for 22KB of data to be allocated, which is approaching 70% of the available on-chip SRAM of an individual Epiphany-III core. This is an excellent code to data ratio for a micro-core architecture, especially considering that this is at a high level of compiler optimisation. Furthermore, considering that the full Olympus runtime library is at least 2KB on the micro-core devices, the overhead of the Olympus abstract machine mnemonics and environments model is only around 1KB on the Epiphany-III when compiled with `-O3` optimisation level.

The figures for the MicroBlaze reflect the use of the floating-point emulation option for the LINPACK benchmark. Unsurprisingly, the code size difference is greater on the MicroBlaze in comparison to the Epiphany-III, at between 2 and 2.6 times larger (for both compiler optimisation levels), due to the increased number of operations generated by the Olympus mnemonics over native C, which is amplified by the floating-point emulation code required for the MicroBlaze LINPACK benchmark.

Considering the difference between the compiler optimisation levels for the Olympus abstract machine code, there is up to a 20% advantage, on the Epiphany-III, in terms of code size in selecting `-Os` over `-O3`. However, for the SPARCv9 the advantage is minimal (0.048%) and is actually detrimental on the ARM32 (-1.67%). Overall, there is an average increase in code size of 7.5% selecting `-O3` over `-Os`, which needs to be considered relative to any runtime performance advantage gained by selecting the higher compiler optimisation level. For a micro-core architecture, such as the Epiphany-III, the code size saving of 20% (approximately 1.8KB) could be sig-

Table 6.9: LINPACK kernel code size (bytes)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	8904	10708	4124	7012
MicroBlaze	14008	15780	5456	7048
PicoRV32	32484	33348	19056	20420
ARM32	14171	13935	5731	5909
MIPS32	23772	24460	8166	9258
AMD64	22111	24279	7376	9929
SPARCv9	22876	22887	8602	9917
RISCV64	15903	16415	5795	6445

nificant. Therefore, it is important to understand any runtime performance differences between the two compiler optimisation levels.

As discussed in Chapter 5, the different benchmarks were chosen to determine if the kernel algorithm impacted the size and runtime performance of Olympus kernels versus those written in native C. The results for the kernel code size comparison between Olympus and native C at optimisation level -O3 for the Jacobi and LINPACK benchmarks have similar ranges, with the Olympus kernels being between 1.06 and 3.8 times the size of native C kernels for the Jacobi benchmark. This indicates that whilst the memory access profiles of different algorithms impacts the size of the Olympus kernels, the range is still relatively constrained between parity (1.06 times) and almost 4 times for both the benchmarks.

6.3.2.6 LINPACK benchmark: kernel runtime performance

The LINPACK benchmark provides a measurement of the processor’s performance by calculating the number of floating point operation per second (FLOPS). Figure 6.9 shows the single-core runtime performance results for LINPACK on the target processor architectures, compiled using the -Os and -O3 compiler optimisation levels. Whilst the results vary widely across the architectures, the runtime performance difference between the Olympus and native C kernels is very small. However, from Table 6.10, it can be calculated that the Olympus LINPACK kernel compiled at -Os is 1.7 times faster than native C on the Epiphany-III and is marginally faster (1.5%) on the ARM32. Although the runtime performance advantage of Olympus kernels on the

⁷On the PicoRV32 at GCC optimisation level -O3, both the Olympus and native C benchmarks abort with a “The matrix A is apparently singular” LINPACK error. As both versions of the benchmark fail, it is suspected that there is a code generation issue for the PicoRV32 at this level of optimisation

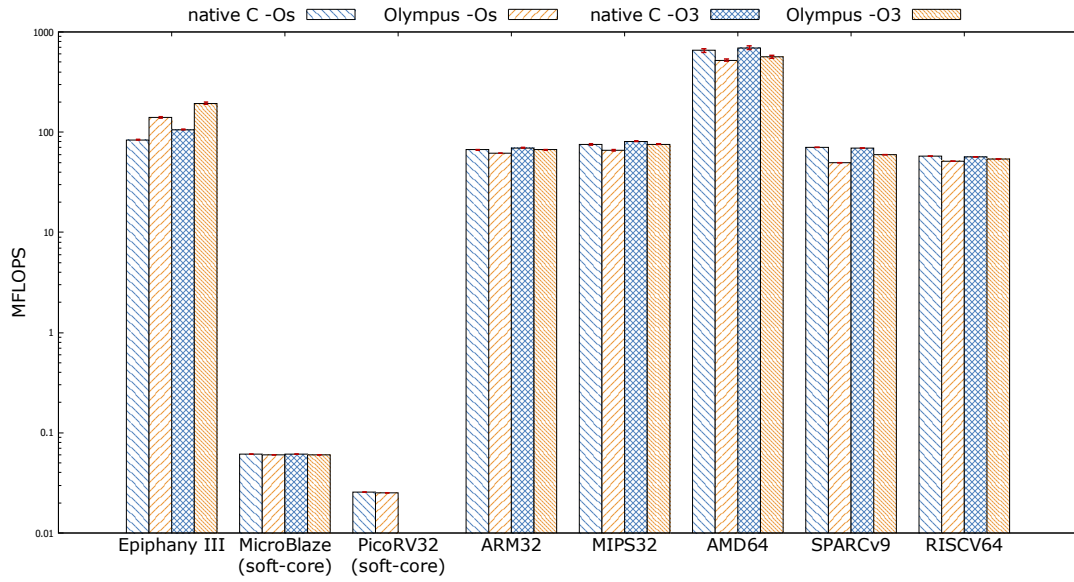


Figure 6.9: Comparing native C and Olympus runtime performance across architectures: LINPACK benchmark. Mean runtime (M) in seconds, with error bars denoting standard deviation (log scale)

Table 6.10: LINPACK kernel mean (M) runtime performance (MFLOPS)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	139.4	193.5	83.99	106.0
MicroBlaze	0.06038	0.06048	0.06113	0.06117
PicoRV32	0.02516	N/A ⁷	0.02563	N/A
ARM32	67.74	66.61	66.71	69.67
MIPS32	65.90	75.60	75.40	81.08
AMD64	521.0	563.0	652.5	695.6
SPARCv9	49.81	59.40	70.78	68.96
RISCv64	51.63	54.00	57.48	56.57

ARM32 is reversed at -O3, where native C is 4.6% faster, the advantage is actually slightly increased at -O3 on the Epiphany-III to 1.8 times faster than native C. On the other architectures, native C is between about 1.2% on the MicroBlaze and 42% on the SPARCv9 faster than Olympus at -Os and between about 7.2% on the MIPS32 and 24% on the AMD64 faster at -O3.

Analysing the runtime performance advantage of Olympus kernels over native C on the Epiphany-III and at -Os on the ARM32 requires knowledge of the peculiarities of the Epiphany-III and looking at the assembly language generated by the C compiler. that requires further investigation.

In the case of the Epiphany-III, there are four modes for the floating point unit (FPU) that can be specified at compile time. From [122]:

- *caller*: Any mode at function entry is valid, and retained or restored when the function returns, and when it calls other functions. This mode is useful for compiling libraries or other compilation units you might want to incorporate into different programs with different prevailing FPU modes, and the convenience of being able to use a single object file outweighs the size and speed overhead for any extra mode switching that might be needed, compared with what would be needed with a more specific choice of prevailing FPU mode.
- *truncate*: This is the mode used for floating point calculations with truncating (i.e. round towards zero) rounding mode. That includes conversion from floating point to integer.
- *round-nearest*: This is the mode used for floating point calculations with round-to-nearest-or-even rounding mode.
- *int*: This is the mode used to perform integer calculations in the FPU, e.g. integer multiply, or integer multiply-and-accumulate.

The default FPU mode is *caller*, which results⁸ in native C kernels being 1.7 times faster than Olympus. The *truncate* FPU mode does not provide a significant improvement (2.1%) of native C kernels over Olympus. The *round-nearest* mode provides a 2.1 times runtime performance improvement of native C over the Olympus abstract machine. As detailed in Table 6.10, the *int* FPU mode, executing integer operations as well as floating point operations in the FPU, delivers a 1.66 and 1.83 times runtime performance advantage of Olympus kernels over native C at for `-Os` and for `-O3`, respectively. This result is surprising but considering that the Epiphany-III is a super-scalar design that can execute two floating point operations and one integer instruction per clock cycle [13], it is possible to surmise that the Olympus mnemonics can take advantage of the additional two integer operations per clock cycle afforded by the *int* FPU mode. For example, the memory addressing requires additional offset calculations for each object reference that could perhaps be performed by the FPU, which coupled with their volume could prevent the pipeline from stalling. However, it is difficult to determine this from the disassembled kernel binary assembly language listing.

⁸The FPU mode comparisons were all performed using the `-O3` compiler optimisation level.

The advantage of the LINPACK benchmark is that it validates and outputs the calculation results, which were compared to validate that the same results were obtained for all FPU mode configurations and the runtime performance advantage of the *int* FPU mode was not due to the kernels exiting early due to a floating point error or a similar issue.

The minor runtime performance advantage of Olympus over native C on the ARM at the `-Os` compiler optimisation level can be explained by the additional 21 `APSR_nzcv` opcodes in the native C kernel. This opcode transfers the floating-point status flags are transferred the ARM application program status register (APSR) and, as [183] state:

“ These instructions stall the ARM until all current NEON or VFP operations complete. ”

It is also interesting to determine from the disassembly listing of the ARM Olympus kernel that the ARM NEON vector / SIMD instructions (e.g. `VLDR`, `VLMUL` and `VSTR`) are being issued by the C compiler for the Olympus mnemonics, thereby taking advantage of this parallel processing capability of the ARM processor for the LINPACK benchmark.

6.3.2.7 Sieve of Eratosthenes benchmark: kernel code size

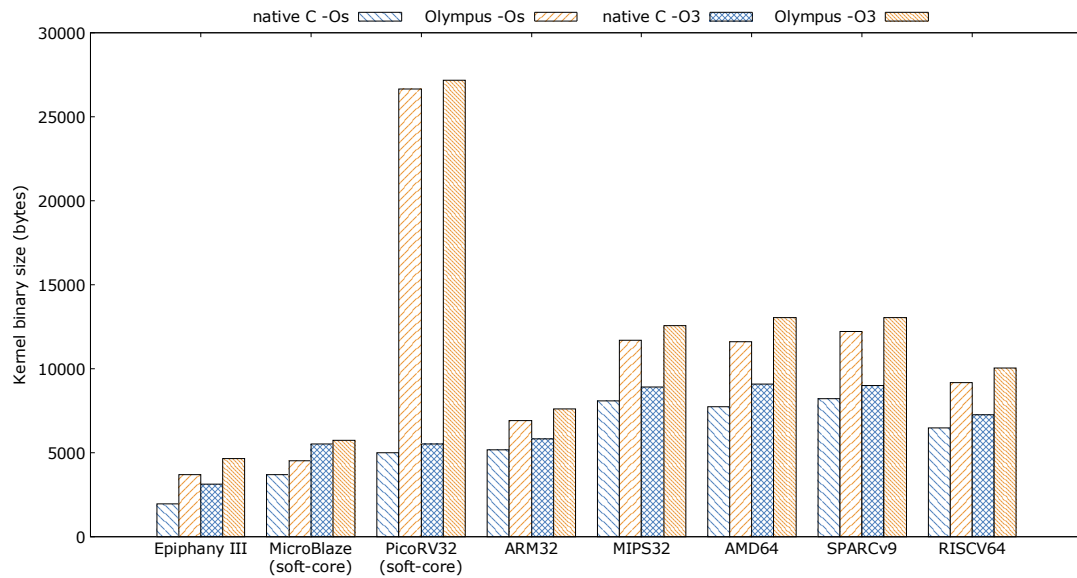


Figure 6.10: Comparing native C and Olympus kernel code size across architectures: Sieve benchmark

As discussed in Section 5.3.3.5, the Jacobi and LINPACK benchmarks test the floating point performance of the Olympus abstract machine. Therefore, the Sieve

of Eratosthenes (Sieve) benchmark was selected to determine the size efficiency and integer performance of Olympus relative to handwritten (native) C. Figure 6.10 shows the size of the Sieve kernels compiled with `-Os` and `-O3` compiler optimisation levels for all CPUs.

Table 6.11: Sieve benchmark code size (bytes)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	3678	4666	1938	3126
MicroBlaze	4520	5724	3708	5516
PicoRV32	26668	27160	4980	5540
ARM32	6894	7602	5178	5826
MIPS32	11701	12582	8097	8894
AMD64	11615	13039	7720	9080
SPARCv9	12195	13059	8231	9019
RISCV64	9185	10025	6470	7264

Whilst, as detailed in Table 6.11, the Olympus kernel sizes are between near parity⁹ and 1.9 times¹⁰ that of the native C kernels for the other CPUs, the difference for the PicoRV32 is striking, with the Olympus kernel size around 5 times larger for both `-Os` and `-O3`. However, when these PicoRV32 Olympus kernel sizes are compared with the Jacobi and LINPACK kernels, the overall size is comparable; the Olympus Sieve kernel size is approximately 27KB, the Jacobi kernel is around 27KB and the LINPACK kernel is about 33KB for compiler optimisation level `-O3`. In comparison, the native C Sieve kernel is 5.5KB, the Jacobi kernel is around 18KB and the LINPACK kernel is 20KB, for the same compiler optimisation level. The Olympus kernel binary size similarity can be explained by the fact that the GCC compiler allocates space¹¹ in the kernel ELF file (Figure 4.13) for the statically allocated C array used for the heap in the Olympus abstract machine. This is best illustrated by the size of the `.bss` segment¹² reported by the GNU `size` utility for the Olympus Sieve kernel on the Epiphany-III, as shown in Listing 6.1, where the Olympus abstract machine heap is 24KB, and the 8MB default heap size of the RISCV64 desktop (threaded) kernel, as shown in Listing 6.2.

⁹MicroBlaze GCC optimisation level `-Os`.

¹⁰Epiphany-III GCC optimisation level `-O3`.

¹¹The resulting binary file size does not increase; when the code is loaded, the heap base (start) address is offset by the required amount.

¹²The Block Started by Symbol (BSS) segment contains the uninitialised objects of an ELF binary. For example, uninitialised global variables, such as the Olympus heap array.

1	text	data	bss	dec	hex filename
2	4666	1208	25336	31210	79ea e_task.elf

Listing 6.1: Output of GNU `size` for Epiphany-III Olympus Sieve kernel

1	text	data	bss	dec	hex filename
2	9185	928	8001440	8011553	7a3f21 threaded_sieve.elf

Listing 6.2: Output of GNU `size` for RISC-V64 Olympus Sieve kernel

1	text	data	bss	dec	hex filename
2	54668	0	0	54668	d58c rv_task.elf

Listing 6.3: Output of GNU `size` for PicoRV32 Olympus Sieve kernel

In contrast, for the PicoRV32, as shown in Listing 6.3, there is only a single `.text` segment, containing the executable code, static values, strings and the Olympus heap array. This is due to the custom GNU *linker file* (Listing E.1 in Appendix E) that is required to set up the memory map on the bare-metal PicoRV32 micro-core. The Epiphany-III and MicroBlaze micro-cores require similar custom linker files. However, the PicoRV32 file is unique in that the `KEEP` [184] command is used to prevent the linker from performing *dead code removal* on the `.text` segment, which is vital to ensure that the PicoRV32 register initialisation is performed. As the register initialisation subroutine (Listing E.2 in Appendix E) is not referenced in the C source code, it would be removed by the GCC linker when the kernel binary is created, if the `KEEP` command was not used. As all functions are placed in the `.text` segment and no dead code removal is performed, all unused library functions will also be kept in the final binary, unlike the binaries for other CPUs. Although this is an issue for PicoRV32 binaries, it impacts both Olympus and native C kernels. Therefore, a more detailed discussion of possible mitigations for this issue will not be provided, except to highlight the benefits of the dynamic code loading mechanism assessed in Section 6.4.

The single `.text` code segment of PicoRV32 binaries obfuscates accurate comparisons of Olympus abstract machine and native C kernel binary sizes for this CPU. The size of the statically allocated C array for the Olympus abstract machine could be subtracted from the overall kernel size to provide an indicative code size for compar-

ison with the corresponding native C kernel but in the case of the Sieve kernel, this results in Olympus kernel sizes of 2092 bytes at `-Os` and 2584 bytes at `-O3`, which are approximately 50% of the size of the equivalent native C kernel binaries. Therefore, code size comparisons between Olympus and native C PicoRV32 kernels need careful analysis. However, direct comparisons can be made for the other CPUs.

In comparison to the LINPACK kernels, the Sieve Olympus kernels are closer in size to the native C kernels, ranging from near parity to 1.9 times for Sieve versus 2 to 3 times for LINPACK. This can be explained by the significantly lower complexity of the Sieve code versus that of LINPACK, where not only will the simpler source code generate less of the wordy Olympus abstract machine mnemonics relative to native C but also the integer-based code will generate less casting operations in comparison to the floating point code of LINPACK. In terms of Olympus kernel size between GCC optimisation levels `-O3` and `-Os`, the results are more comparable at 27% for Sieve and 20% for LINPACK. The 7% size reduction provided by `-Os` over `-O3` for the Sieve Olympus kernels suggests that the Olympus integer memory access code can be optimised for size more fully by GCC than the floating point memory access code (at `-O3`), accepting that the floating point will still require additional casting operations, irrespective of the processes performed by the compiler to reduce the binary size.

6.3.2.8 Sieve of Eratosthenes benchmark: kernel runtime performance

Figure 6.11 shows that, across compiler optimisation levels `-Os` and `-O3`, the Sieve benchmark displays a wider runtime performance gap between the Olympus and native C kernels than was observed for the Jacobi and LINPACK benchmarks, discussed in Sections 6.3.2.4 and 6.3.2.6, respectively. From Table 6.12, the Olympus Sieve kernel runtime performance ranges from approximately 1.4 times slower than native C on the Epiphany-III to over 5.5 times slower¹³ on the RISC-V64. For all CPUs apart from the AMD64, the difference between Olympus and native C kernel runtime performance is smaller at compiler optimisation level `-O3` than at `-Os`. On the RISC-V64, the native C Sieve kernel is 5.5 times faster than the Olympus kernel at `-Os` but is only 4 times faster at `-O3`. Whilst the kernel runtime performance difference between the `-Os` and `-O3` GCC optimisation levels is greatest for the RISC-V64, all of the RISC CPU Olympus kernels close the runtime performance gap with the native C kernels at `-O3`. In comparison, the CISC AMD64 (x86-64) native C kernels are 1.7 times faster than

¹³Both bounds of the range at compiler optimisation level `-O3`.

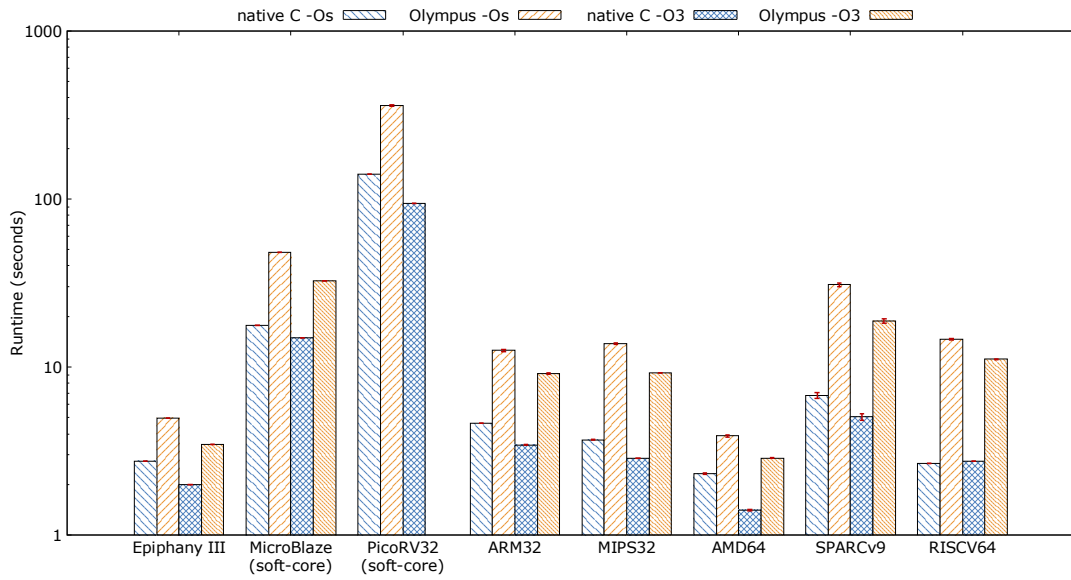


Figure 6.11: Comparing native C and Olympus runtime performance across architectures: Sieve benchmark. Mean runtime (M) in seconds, with error bars denoting standard deviation (log scale)

Olympus at $-Os$ and 2 times faster at $-O3$. This suggests that GCC is able to leverage the additional registers available on the RISCv64 (32 registers [164]) over those available on the AMD64 (16 registers [164]) to optimise the Olympus abstract machine code at $-O3$ optimisation level. However, the results for the Epiphany-III, MIPS32 and SPARCv9 suggest that the additional registers available on the Epiphany-III (64 registers) [122] do not provide an advantage over the 32 available on the MIPS32 and SPARCv9 [33]. The approximate 3% narrowing of the runtime performance gap between native C and Olympus kernels at $-O3$ over $-Os$ on the Epiphany-III, is significantly less than the 13% and 20% narrowing of the runtime performance gap at $-O3$ on the MIPS32 and SPARCv9, respectively.

The results of the Sieve benchmark when compared to those obtained for Jacobi and LINPACK suggest that the Olympus abstract machine is sub-optimal for integer operations. It is possible to conjecture that the runtime performance overhead of the Olympus Sieve kernels is due to the fact that GCC is able to optimise the memory access of the C kernel byte array elements, where an integer, rather than a character, is read from memory at a time¹⁴, thereby reducing the total number of memory accesses in comparison to the Olympus kernel. However, the runtime performance figures for

¹⁴Four bytes on the Epiphany-III, MicroBlaze, PicoRV32 and ARM32, and eight bytes on the AMD64, SPARCv9 and RISCv64.

the AMD64 do not reflect this hypothesis, as the native C kernel is only 1.7 times faster than the Olympus kernel at `-Os` and about 2 times faster at `-O3`.

Table 6.12: Sieve benchmark native C and Olympus kernel mean (M) runtime (seconds)

CPU	Olympus (-Os)	Olympus (-O3)	native C (-Os)	native C (-O3)
Epiphany-III	4.972	3.475	2.760	1.996
MicroBlaze	48.23	32.70	17.78	14.94
PicoRV32	358.6	N/A ¹⁵	140.8	94.52
ARM32	12.53	9.135	4.629	3.443
MIPS32	13.79	9.211	3.683	2.858
AMD64	3.895	2.868	2.319	1.409
SPARCv9	30.93	18.80	6.763	5.060
RISCV64	14.61	11.12	2.676	2.755

On the PicoRV32, the Olympus `-O3` kernel froze and did not return a value to the host, even though the kernel successfully executed when compiled at optimisation level `-Os`. Furthermore, although the GCC documentation [180] states that optimisation level `-Os` enables all of the `-O2` level optimisations except those listed in Listing 6.4, the PicoRV32 Olympus Sieve kernel also froze when compiled for the PicoRV32 at `-O2`. The kernels were found to successfully execute at `-Os` plus the `-falign-functions`, `-falign-jumps`, `-falign-loops` and `-falign-labels` options. Analysis of the options that comprise the GCC `-O3` optimisation level determined that the `-fdevirtualize-speculatively` option applied to the previous `-Os` plus additional options configuration caused the Olympus Sieve kernels to freeze. The `-fdevirtualize-speculatively` option description from [180] states:

“ Attempt to convert calls to virtual functions to speculative direct calls... [C]hange the call into a conditional deciding between direct and indirect calls. The speculative calls enable more optimizations, such as inlining. ”

It is possible to speculate that, although no C++ code forms part of the Olympus abstract machine or the Eithne framework, the function pointers used to dispatch functions within the abstract machine and the Eithne framework are being incorrectly *devirtualized* by GCC, leading to invalid optimisation that results in the kernels freezing. As both the Olympus and native C LINPACK kernels freeze at optimisation level `-O3` on the PicoRV32 and the Eithne framework uses function pointers to dispatch kernel functions, it is possible to infer that the GCC issue is located within the generation of the Eithne kernel launch code, rather than that for the Olympus abstract machine.

¹⁵The Olympus Sieve kernel freezes at compiler optimisation level `-O3`.

```

1 -falign-functions -falign-jumps -falign-loops
2 -falign-labels -freorder-blocks -freorder-blocks-algorithm=stc
3 -freorder-blocks-and-partition -fprefetch-loop-arrays

```

Listing 6.4: `-O2` optimisation flags disabled at `-Os` [180]

Table 6.12 highlights an anomaly for the native C Sieve kernel runtime performance on the RISC-V64, where the `-Os` kernel is marginally (0.079 seconds) faster than the `-O3` kernel, yet the Olympus `-O3` kernel is 24% (3.49 seconds) faster than the `-Os` kernel. However, the difference is small enough to be considered insignificant to the findings of the benchmark results.

6.4 Managing arbitrary code size (RQ2)

To validate the benefits of the Olympus dynamic binding and code loading approach in terms of runtime performance and kernel size, an updated version of the Jacobi benchmark was developed that used a modified version of the original Olympus compiler and abstract machine to create function calls to preserve a programming language’s *block structure* [146], whereby each Python `while` loop body is wrapped in a generated C function that is then called for each loop iteration. This option was originally used as the default for the Olympus compiler since declaring and calling a function implicitly creates a new scope level (Olympus stack frame) within which new variables are declared. The runtime performance impact is small as the loop bodies are *inlined* by the C compiler at optimisation level `-O3`, removing the cost of the function call, leaving only the overhead of creating the desired Olympus stack frame. The Olympus ePython compiler no longer uses this approach as Python does not support block structured loop constructs. However, a simple modification to remove the stack frame creation and mark the generated functions for dynamic loading, provided a mechanism to generate C source code that could then be modified to support the evaluation of the code size and runtime performance of the following function dispatch models¹⁶:

- *Static dispatch*: functions are statically bound to the executable and optimised for direct execution

¹⁶These synthetically created kernel variations were also deployed and timed using the Eithne benchmarking framework.

- *Dynamic dispatch*: functions are statically bound to the executable but dynamically dispatched via lookup in the environment
- *Dynamic binding*: functions are dynamically loaded by the kernel and dispatched via lookup in the environment

These function dispatch models were chosen to confirm the hypothesis that the dynamic dispatch model, required by dynamic binding / function loading introduces a runtime performance overhead in comparison to the static dispatch model where the C compiler is able to optimise the function call, or even inline the code. For statically bound kernels, the Olympus compiler can directly call the generated C functions if they are non-recursive and do not declare any local variables, as they do not need an Olympus stack frame. However, for most functions, dynamic dispatch is used which creates the required stack frame, looks up the function in the environment and applies it. Listing 6.5 shows the the dynamic function dispatch call for the `initialise()` function in the Olympus Jacobi kernel. The Olympus stack frame is created by `MKBETA`, the `LDL` macro references the `initialise()` function code referenced at offset 15 in the frame, the two `PARAM` macros add the `u_k` and `u_kp1` arrays as arguments and the `APPLYV` macro applies the void function. As dynamically bound functions need to be dynamically dispatched, as they are *linked* to the kernel at runtime, it is crucial to understand the runtime performance overhead of this dispatch model.

```

1      (MKBETA (LDL (ADDR (15)) ,
2          PARAM (0, "u_k", LDA (ADDR (1)) ) ,
3          PARAM (1, "u_kp1", LDA (ADDR (2)) ) ,
4          APPLYV) ;

```

Listing 6.5: Olympus dynamic dispatch function call example

To compare the lookup and dispatch overhead, the compiler was modified to generate both dynamically¹⁷ and statically dispatched function calls within the generated C file, with the dispatch model selected by a C compiler `#define` directive, as shown in Listing 6.6. This code could then be compiled manually and deployed to the microcores using the Eithne framework.

¹⁷As discussed, this dynamic dispatch has been modified to remove the standard `MKBETA` call to create the stack frame.

```

1 while (LDI (ADDR1 (7)) <= LDI (ADDR1 (0))) {
2   #ifdef DYNAMIC_DISPATCH
3     (LDL (ADDR1 (11)),
4     APPLYV);
5   #else
6     oly_v$a1$v (env, self);
7   #endif
8   INCI (ADDR1 (7));
9 }

```

Listing 6.6: Olympus static and dynamic function call example

Listing 6.7 shows the generated dynamic function loading calls in the dynamically bound Olympus Jacobi kernel, using the `DYNLDF` API to retrieve an Olympus generated function from the host. As the Jacobi kernel also requires the native C `powf()` and `sqrtf()` functions, these are loaded by the `eithne_get_function()` calls.

```

1 DECLL (11, "oly_a1", DYNLDF ("oly_v$a1$v", 0));
2 DECLL (12, "oly_a2", DYNLDF ("oly_v$a2$v", 0));
3 DECLL (13, "oly_a3", DYNLDF ("oly_v$a3$v", 0));
4 DECLL (14, "oly_a4", DYNLDF ("oly_v$a4$v", 0));
5 DECLL (15, "initialise", DYNLDF ("oly_v$initialise$RR", 3));
6 d_powf=eithne_get_function("d_powf");
7 d_sqrt=eithne_get_function("d_sqrt");

```

Listing 6.7: Olympus Jacobi kernel dynamic binding and loading

6.4.1 Jacobi benchmark: dynamically loaded code size

Before evaluating the different function dispatch models, the same kernels were used to compare the code size for statically and dynamically bound Jacobi kernels on the target micro-cores (Epiphany-III, MicroBlaze and PicoRV32), compiled with the GCC compiler `-O0` and `-O3` optimisation levels, as shown in Figure 6.12. As expected, the dynamically bound kernels are smaller than the statically bound versions for both `-O0` and `-O3` compiler optimisation levels. Again, as expected, the kernels compiled with GCC `-O0` are smaller than those compiled with `-O3` optimisation. For the Epiphany-III and MicroBlaze with hardware FPU, the differences between the static and dynamic kernels is small, as the size of the dynamically loaded functions are relatively small in comparison to the overall kernel runtime support code. However, the differences

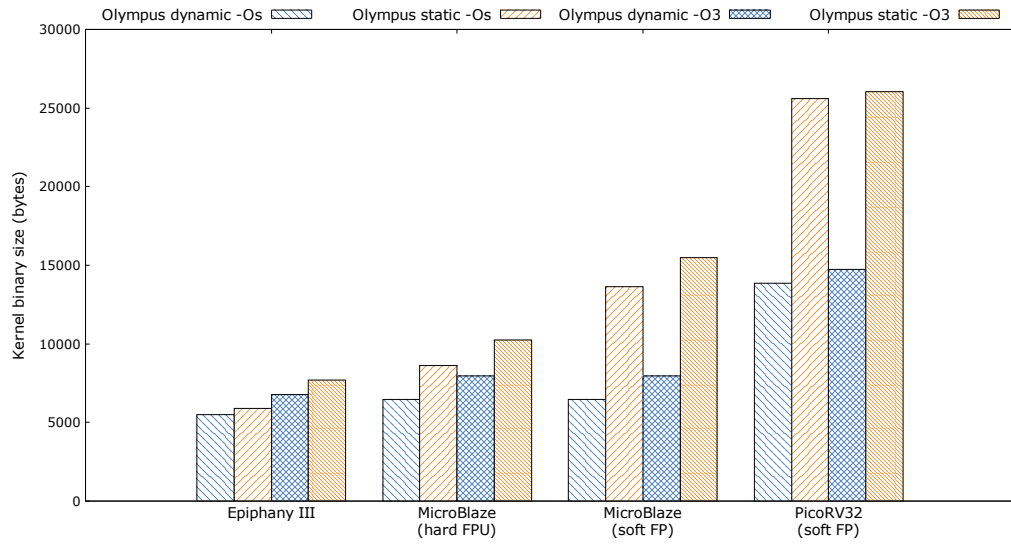


Figure 6.12: Comparing static and dynamic Olympus kernel size across architectures: Jacobi benchmark

are much more marked for the software floating point versions of the kernels for both the MicroBlaze and PicoRV32. For these micro-cores, the additional code required to provide floating point emulation required by the kernel functions is significant. As Table 6.13 details, the statically bound kernel for the MicroBlaze is between 49% and 53% bigger than the dynamically bound kernel for the `-O3` and `-Os` compiler optimisation levels, respectively. Similar results are obtained for the PicoRV32, where dynamic binding provides a saving of between 44% and 46% in initial kernel code size. Whilst the kernels still require the dynamically bound functions to be loaded into memory before execution, which will add to the overall memory footprint, these can be unloaded after use to save memory. There are clear benefits to dynamically loading and unloading kernel functions that require floating point support, considering that the dynamically bound kernels for both the hard FPU and soft floating point emulation on the MicroBlaze are identical for each compiler optimisation level, indicating that all floating point operations are within the dynamically bound functions.

Table 6.13: Static and dynamic Olympus Jacobi kernel size (bytes)

Micro-core	Dynamic -Os	Dynamic -O3	Static -Os	Static -O3
Epiphany-III	5498	6782	5914	7686
MicroBlaze (hard FPU)	6468	7972	8632	10228
MicroBlaze (soft FP)	6468	7972	13636	15504
PicoRV32 (soft FP)	13850	14732	25600	26056

Table 6.14: Epiphany-III modified Jacobi kernel code size (bytes) for ePython VM, Olympus dispatch models and native C

Variant	Runtime	Bytecode	Functions	Total
ePython VM	21522	2329	-	23851
Static dispatch	9810	-	-	9810
Dynamic dispatch	12162	-	-	12162
Dynamic loading	6774	-	1088	7862
Native C	7398	-	-	7398

A comparison of the same Python version of the Jacobi benchmark running on the ePython VM is provided in Table 6.14, which details the overall code size for the different Jacobi benchmark variants, with the VM size including the interpreter, runtime support and bytecode, and the Olympus dynamic loading including the static binary (kernel and runtime support), plus the dynamically loaded kernel functions. The non-applicable entries in the table are marked with a dash; for example, bytecode is not applicable for the Olympus and native C variants. As expected, for the overall deployed binary size, the native C variant of the code is the smallest, with dynamically loaded Olympus variant trailing by only 464 bytes. This is followed by the Olympus variant with all functions statically bound and dispatched at 9810 bytes, where the compiler is able to optimise function calls. The dynamic dispatch variant is 24% bigger, as the dynamic dispatch method requires more code (function lookups in the environment) and prevents the C compiler from optimising the function calls via inlining. Finally, the ePython VM variant of the benchmark has the smallest compiled code (bytecode) size but requires the VM of around 22KB to execute.

It should be noted that the static dispatch option can only be used for very simple codes, where the Olympus compiler can detect simple function calls that do not need new frames, for example, those without local variables. Therefore, for most functions the compiler will generate code that uses the dynamic dispatch model. Whilst this option produces code that is approximately 64% larger than native C overall, it uses only 51% of the overall memory required by the ePython VM variant and has an approximately 30% smaller initial memory footprint than the native C kernel. Therefore, for the same Python source code, the Olympus dynamically loaded kernel functions can provide significantly more of the micro-cores' limited on-core memory is available for data.

6.4.2 Jacobi benchmark: dynamic loaded code runtime performance

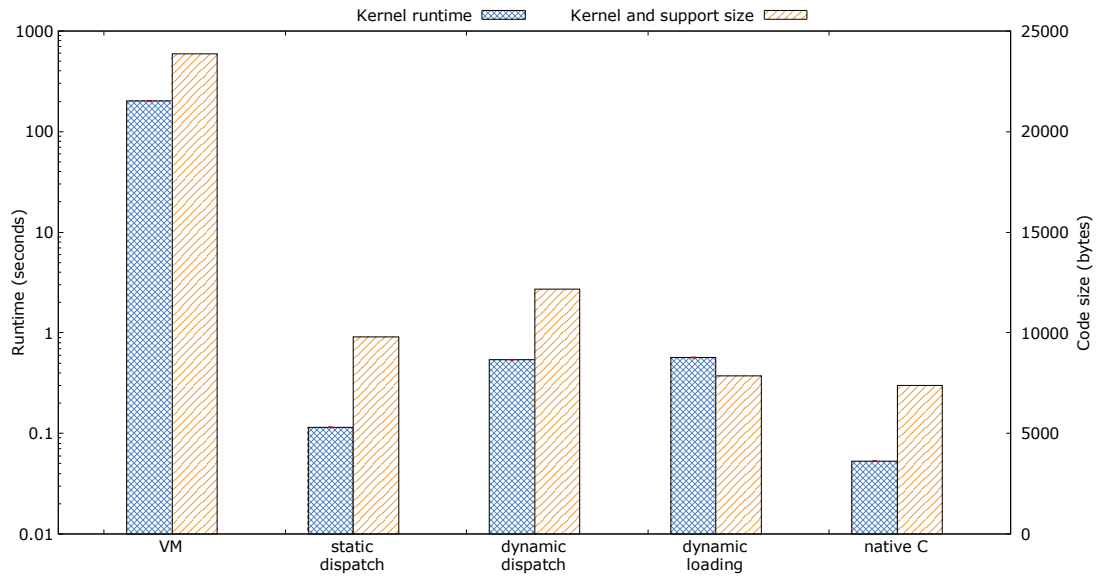


Figure 6.13: Relative mean (M) kernel runtime performance and binary code size for the modified Jacobi benchmark on the Adapteva Epiphany-III. Left y-axis (log scale) is the kernel runtime in seconds (blue) and right y-axis is the kernel and support code size in bytes (orange)

Figure 6.13 shows the relative code sizes and runtime performance for all the benchmark variants. The native C benchmark kernel has the fastest execution time at 0.053 seconds and the static dispatch code generation variant is round 2 times slower at 0.115 seconds. The dynamic dispatch model decreases runtime performance relative to the static dispatch model by approximately 5 times at 0.539 seconds and the dynamic loading version is only marginally slower at 0.566 seconds. As shown in Table 6.15, the default dynamic function dispatch code generation version is significantly faster than the VM version, which requires around 201 seconds to execute the same Python kernel. However, it should be noted that ePython is also executing the bytecode from significantly slower off-chip memory (150 MB/s maximum bandwidth obtainable in practice [185]), as the kernel bytecode and heap requirements are too large to allow the bytecode to execute from on-chip RAM. Considering that code of the dynamic loading version also only requires approximately a third of the memory to execute, not only is there a significant runtime performance increase over the VM, but also the ability to handle much more data at the same time, as well as being able to execute arbitrary sized codes via dynamic function loading. This greatly increases the

practical applications that dynamic languages can support on micro-core architectures.

Table 6.15: Epiphany-III modified Jacobi kernel mean (M) runtimes (seconds), with associated standard deviation (SD), for ePython VM, Olympus dispatch models and native C

Variant	M runtime	SD
ePython VM	201.54	0.2189
Static dispatch	0.1149	0.00001
Dynamic dispatch	0.5390	0.00066
Dynamic loading	0.5658	0.00062
Native C	0.0526	0.00001

The synthetic Jacobi benchmark uses a modified version of the original Olympus compiler and abstract machine, where the runtime performance overhead over native C is higher than that for the LINPACK and Sieve of Eratosthenes benchmarks, which use the latest version of the compiler and abstract machine. The abstract machine has undergone a number of iterations and refinements during the research to provide greater runtime performance, particularly with respect to memory addressing and stack frame management. A detailed description of the current Olympus abstract machine can be found in Sections 4.3.3.4 and 4.3.3.5. It is important to note that the Olympus LINPACK and Sieve of Eratosthenes benchmark kernels use dynamic function dispatch and the runtime performance of the Olympus LINPACK kernels can match, or even exceed, that of the native C kernels and are, at worst, 5.5 times slower for the Sieve benchmark, in comparison to the factor of 10 slow down exhibited for the synthetic Jacobi dynamic loading benchmark. Therefore, the development of the Olympus abstract machine over time has significantly reduced the runtime performance gap with native C, even for kernels with the default dynamic function dispatch.

6.5 Summative Olympus / native C analysis

6.5.1 Summative analysis: kernel code size

Table 6.16 details the Olympus to native C kernel binary size ratios for the Fibonacci, Jacobi, LINPACK and Byte Sieve benchmarks for all CPUs outlined in Section 5.3.1, compiler optimisation level $-O3$. Table 6.16 also provides the geometric means (GM) for all benchmarks per CPU and all CPUs per benchmark. The smallest Olympus

Table 6.16: Kernel binary size ratios between Olympus and native C at -O3, including geometric means (*GM*) of all benchmarks per CPU and all CPUs per benchmark.

CPU	Fibonacci	Jacobi	LINPACK	Byte Sieve	<i>GM</i>
Epiphany-III	1.621	1.334	1.527	1.493	1.490
MicroBlaze	1.112	1.067	1.700	1.038	1.203
PicoRV32	2.309	1.010	1.232	1.945	1.538
ARM32	1.858	1.349	2.358	1.305	1.666
MIPS32	1.892	1.264	2.642	1.415	1.729
AMD64	1.931	3.860	2.445	1.436	2.262
SPARCv9	1.818	1.356	2.308	1.448	1.694
RISCV64	1.800	1.318	2.547	1.380	1.699
<i>GM</i>	1.761	1.424	2.028	1.414	

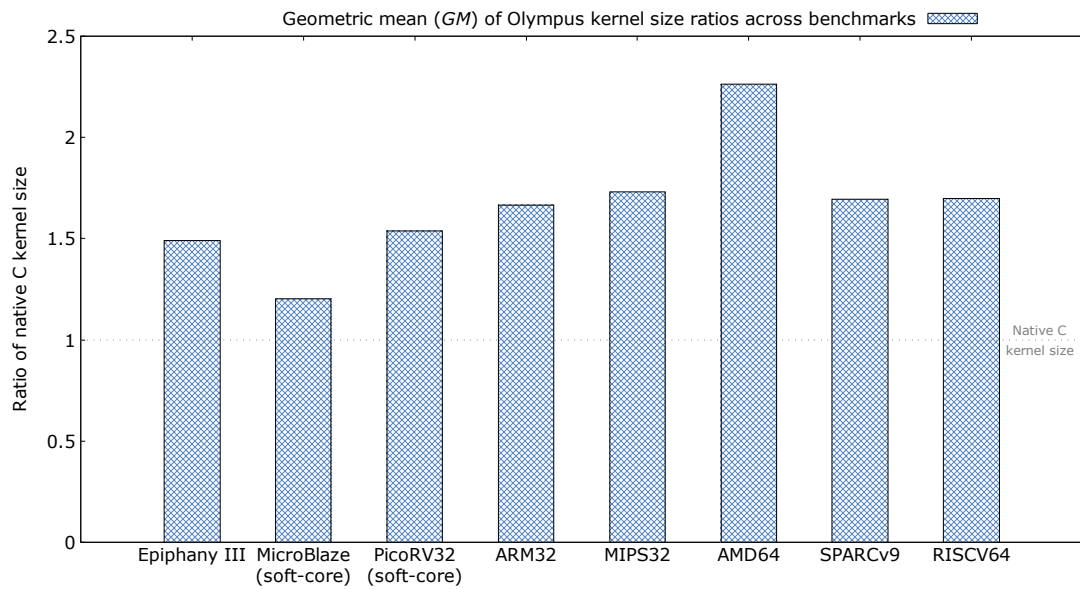


Figure 6.14: Geometric mean of the ratio of Olympus to native C kernel binary size for all benchmarks per CPU, where native C kernel binary size equals 1

kernel binary size ratio is approximately parity with native C, obtained by the Jacobi benchmark on the PicoRV32, yet the same benchmark yields the highest ratio of around four times that of native C on the AMD64. The PicoRV32 result can be explained by the additional code required to support software floating point emulation relative to the kernel code size, for both the Olympus and native C kernels. This is supported by the PicoRV32 LINPACK Olympus code size ratio of around 1.2 times that of native C and the approximate two times ratio of both the integer-only Fibonacci and Byte Sieve kernels. From the perspective of the Jacobi code on the AMD64, the GCC compiler

appears particularly adept at optimising the C code for both size and performance, in comparison to the Olympus kernel code.

Figure 6.14 illustrates that across all benchmarks, the Olympus kernel binary size ratio is highest on the AMD64, further supporting the assertion that the Olympus abstract machine code is *wordy*, with associated possible additional *register spillage* code on this CPU¹⁸, in comparison to the native C kernel code. However, as Table 6.18 highlights, this does not appear to unduly impact performance on the AMD64 ($GM = 75.34\%$ of native C performance).

Table 6.16 highlights that the average of Olympus benchmark kernel sizes on the ARM32, MIPS32, SPARCv9 and RISCv64 are all around 1.7 ($GM = 1.666$ to $GM = 1.729$) times bigger than native C, with the smallest average size ratio across the benchmarks being attained by the MicroBlaze ($GM = 1.203$). As Figure 6.15 shows, the LINPACK benchmark kernels have the highest average code size ratio ($GM = 2.028$) across the CPUs, followed by the Fibonacci ($GM = 1.761$), Jacobi ($GM = 1.424$) and Byte Sieve benchmarks ($GM = 1.414$).

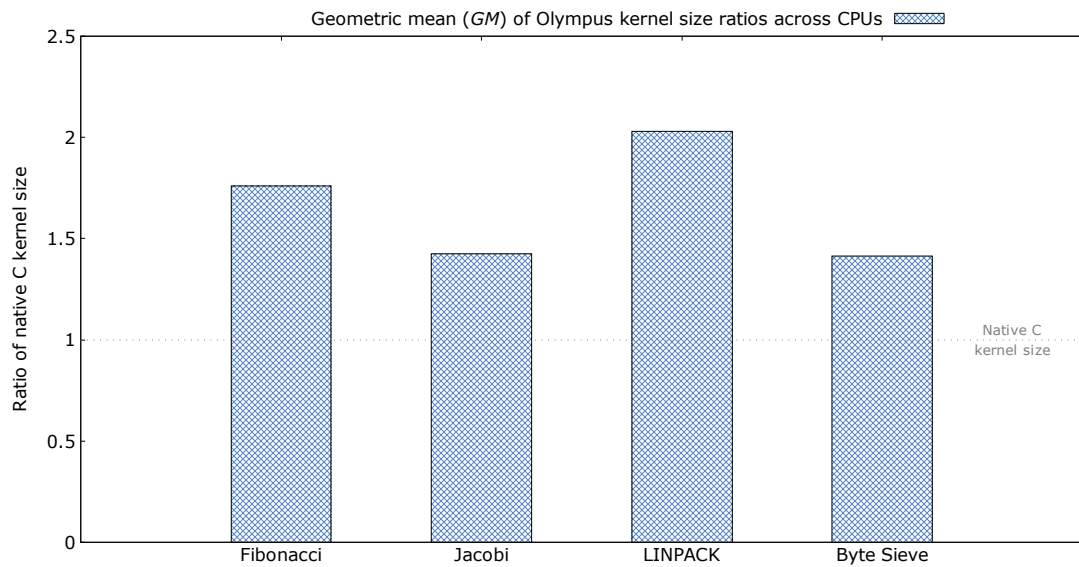


Figure 6.15: Geometric mean of the ratio of Olympus to native C kernel binary size for all CPUs per benchmark, where native C kernel binary size equals 1

¹⁸The AMD64 (x86-64) has 16 registers in comparison to the 32 (64 for the Epiphany-III) of the other CPUs.

6.5.2 Summative analysis: kernel runtime performance

Table 6.17 details the maximum runtime performance attained by the Olympus abstract machine kernels in comparison to native C, for the Fibonacci, Jacobi, LINPACK and Byte Sieve benchmarks for all CPUs listed in Section 5.3.1, at GCC compiler optimisation level $-O3$. As mentioned in Sections 6.3.2.6 and 6.3.2.8, values for the LINPACK and Byte Sieve benchmarks on the PicoRV32 are not available as the RISC-V 32-bit GCC compiler 8.2.0 produces invalid code that cannot be executed.

Table 6.17: Maximum runtime performance ratio between Olympus and native C at $-O3$

CPU	Fibonacci	Jacobi	LINPACK	Byte Sieve
Epiphany-III	101.0%	49.11%	196.2%	57.71%
MicroBlaze	100.2%	71.94%	98.95%	45.77%
PicoRV32	121.7%	97.64%	N/A ¹⁹	N/A
ARM32	100.3%	78.84%	97.69%	39.89%
MIPS32	103.2%	78.19%	95.13%	31.46%
AMD64	133.3%	100.0%	100.0%	51.38%
SPARCv9	129.9%	182.8%	86.55%	32.98%
RISCV64	75.81%	32.17%	95.90%	25.20%

The maximum Olympus runtime performance, at 196.2% of native C, is obtained by the LINPACK benchmark on the Epiphany-III, where it is feasible to surmise that the Olympus mnemonics are able to leverage the additional two integer operations per clock cycle of the Epiphany-III FPU *int* mode to prevent the pipeline from stalling, in comparison to the native C code (Section 6.3.2.6). The second highest runtime performance ratio achieved by Olympus, at 182.8% of native C, was for the Jacobi benchmark on the SPARCv9. However, when the minimum and maximum runtime figures for both Olympus and native C Jacobi kernels are compared on the SPARCv9, the minimum runtimes are identical and the maximum runtimes differ by 0.02s, suggesting that the results are I/O bound, as supported by the mean ratio of $M = 100.4\%$, detailed in Table 6.18.

As listed in Table 6.18 and shown in Figure 6.16, the Olympus geometric mean (*GM*) runtime ratios for all benchmarks per CPU are below 100% except for the PicoRV32, where the lack of data for the LINPACK and Byte Sieve benchmarks, coupled with the 121.7% ratio of the Fibonacci benchmark, skew the Olympus *GM* ratio value

¹⁹GCC 8.2.0 for RISC-V (rv32imc) produces invalid code for both Olympus and native C versions of the LINPACK and Byte Sieve benchmarks at optimisation level $-O3$.

Table 6.18: Mean (M) Olympus runtime performance versus native C at -O3, including geometric means (GM) of all benchmarks per CPU and all CPUs per benchmark.

CPU	Fibonacci	Jacobi	LINPACK	Byte Sieve	GM
Epiphany-III	100.0%	48.66%	182.6%	57.45%	84.52%
MicroBlaze	100.0%	71.94%	98.86%	45.68%	75.50%
PicoRV32	121.7%	97.60%	N/A	N/A	109.0%
ARM32	99.99%	70.75%	95.62%	37.69%	71.06%
MIPS32	100.1%	77.45%	93.24%	31.03%	68.81%
AMD64	100.5%	80.65%	80.95%	49.13%	75.34%
SPARCv9	92.19%	100.4%	86.14%	26.92%	68.07%
RISCV64	95.85%	31.86%	95.45%	24.79%	51.85%
GM	101.0%	68.47%	101.0%	37.31%	

on this processor. When this value is set aside, the Epiphany-III has the highest average runtime ratio ($GM = 84.52\%$). Whilst again, this value is impacted by the high LINPACK value ($M = 182.6\%$) on this processor, the Byte Sieve value ($M = 57.45\%$) is the highest of all the CPUs, with both the the SPARCv9 ($M = 26.92\%$) and RISCV64 ($M = 24.79\%$) attaining less than a third of native C runtime performance. Overall, average Olympus runtime ratio ($GM = 37.31\%$) for the Byte Sieve benchmark across all CPUs suggests that the Olympus abstract machine is poor at handling simple integer array codes like Byte Sieve. However, initial investigations on the RISCV64 [26] suggest that this poor runtime performance in comparison to native C may be due to the use of a `while` loop in the ePython code. An ePython `for` loop uses an internal C loop index variable, whereas the `while` loop uses a user-defined ePython (Olympus) variable for the loop index. The `for` loop native C index variable not only controls the loop iteration but is also used to update the ePython list element; in contrast, the `while` loop requires a lookup of the index variable in the Olympus environment for both loop control and list element updates, impacting runtime performance. The use of an ePython `for` loop for the Byte Sieve benchmark code can result in the `for` loop kernel being 3 times faster than the `while` loop version at -O3 [26].

Figure 6.16 shows that for the MicroBlaze, ARM32, MIPS32 and RISCV64 the maximum Olympus runtime performance figures are approximately equal to native C (the runtime ratios are around 100%). The Epiphany-III, PicoRV32, AMD64 and SPARCv9 maximum runtime performance ratios all exceed native C, with the significantly higher Epiphany-III and SPARCv9 figures ($M = 196.2\%$ and $M = 182.8\%$, respectively), as discussed earlier in this section. Figure 6.16 also shows that the geo-

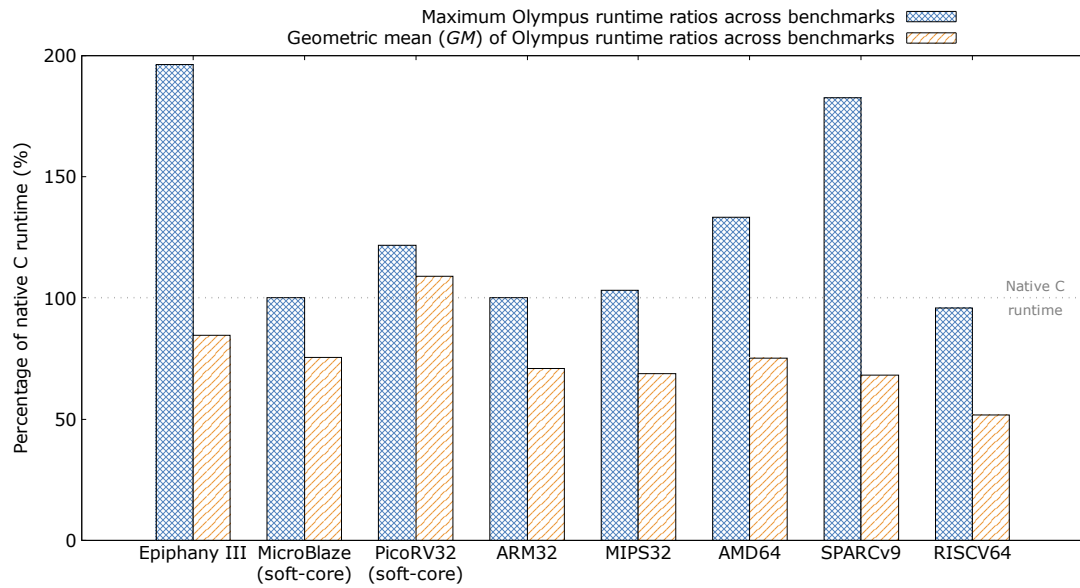


Figure 6.16: Maximum and geometric mean (*GM*) of the ratio of Olympus to native C kernel runtime for all benchmarks per CPU, where native C kernel runtime equals 100%

metric mean of Olympus runtime ratios for the Fibonacci, Jacobi, LINPACK and Byte Sieve benchmarks per CPU are all below native C (100%) except for the PicoRV32, which is marginally higher ($GM = 109.0\%$) due to the lack of values for LINPACK and Byte Sieve, coupled with the high average ratio for the Fibonacci benchmark ($M = 127.7\%$).

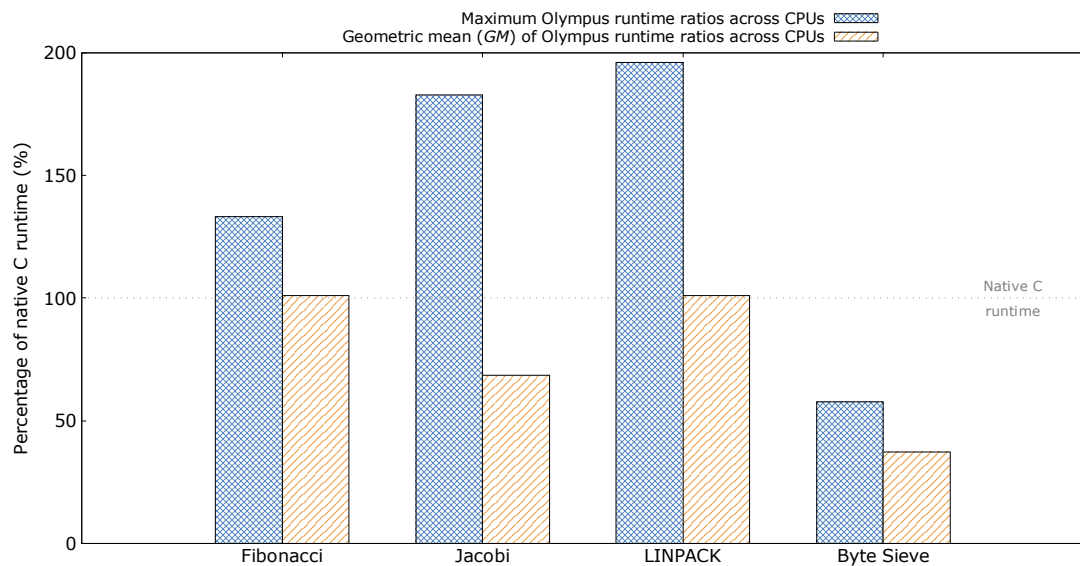


Figure 6.17: Maximum and geometric mean of the ratio of Olympus to native C kernel runtime for all CPUs per benchmark, where native C kernel runtime equals 100%

Figure 6.17 shows that the maximum Olympus runtime ratios for the Fibonacci, Jacobi and LINPACK benchmarks are above native C, with only the Olympus runtime ratio for Byte Sieve being significantly lower at 57.71%. The average Olympus runtime ratio for Byte Sieve ($GM = 37.31\%$) across the CPU architectures is also the lowest of all the benchmarks. The ratios for the Fibonacci and LINPACK benchmarks are comparable to native C (both $GM = 101.0\%$), and the Jacobi benchmark achieves an average across the CPUs of almost 70% of native C performance ($GM = 68.47\%$).

6.6 Desktop comparison: LINPACK benchmark

This work is focused on techniques to manage the limited micro-core on-chip memory for data (RQ1) and code (RQ2), as well as addressing the low performance of virtual machines (RQ3) for dynamic languages on these architectures. As the comparable dynamic language compilation technologies discussed in Chapter 2 do not target micro-core devices, the runtime performance and code size of ePython kernels generated for the Olympus abstract machine have been compared to native C. However, to set these results in context to the analogous desktop compilation technologies, a comparison between the Python interpreter (CPython), Cython, Numba JIT, Olympus and native C for the LINPACK benchmark on an Intel i5 (macOS) was performed. Table 6.19 lists the versions of Python, Cython and Numba used for the comparison and the ePython and native C LINPACK source listings can be found in Appendix B. The LINPACK benchmark results (problem size $n = 1000$) for each technology were averaged (M) over 100 execution runs.

Table 6.19: Desktop Intel i5 (macOS) interpreter and compiler versions

Technology	Version
Python (CPython)	3.9.13
Cython	0.29.30
Numba	0.55.2
Clang	13.1.6

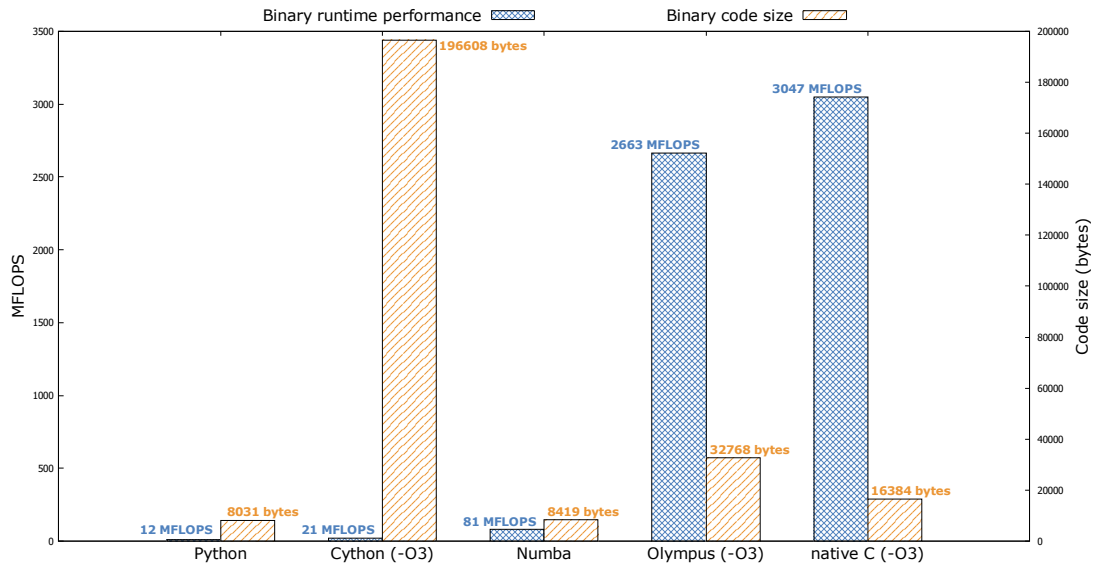


Figure 6.18: Comparing Olympus mean (M) runtime performance (MFLOPS) and binary code size (bytes) with Python, Cython, Numba and native C on the Intel i5. Left y-axis is the kernel floating-point performance in MFLOPS (blue) and right y-axis is the kernel binary code size in bytes (orange)

6.6.1 LINPACK benchmark: code size

Table 6.20 details the LINPACK benchmark binary code size²⁰ for each technology. The Cython, Olympus and native C codes were compiled using Clang at optimisation level `-O3`. The Olympus binary is around 4 times bigger than the Python and Numba bytecode. However, the Olympus binary includes all the runtime support required to execute the code, whereas Python and Numba require the interpreter, which is approximately 3MB for version 3.9.13 on macOS. This results in the overall binary memory footprint of CPython and Numba being over 91 times bigger than that of Olympus. Cython is 6 times larger than Olympus but it is important to note that the Cython code includes all the functions and objects required to support Python codes, thereby replacing the interpreter. As shown in Figure 6.18, the Olympus binary is twice as large as the native C binary, which is the same as the difference between Olympus and native C LINPACK kernels compiled using GCC for the Epiphany-III (Section 6.3.2.5).

²⁰This is the bytecode size for CPython and Numba.

Table 6.20: LINPACK benchmark binary size on Intel i5 (macOS) for Python (CPython), Cython, Numba, Olympus and native C

Technology	bytes
Python (CPython)	8031
Cython (-O3)	196608
Numba (JIT)	8419
Olympus (-O3)	32768
Native C (-O3)	16384

6.6.2 LINPACK benchmark: runtime performance

Table 6.21 details the floating point performance of the LINPACK benchmark in MFLOPS for Python, Cython, Numba, Olympus and native C. As shown in Figure 6.18, Olympus and native C provide significantly greater MFLOPS performance than Python, Cython and Numba; Olympus compiled at optimisation level `-O3` is 218 times faster than Python, is almost 130 times faster than Cython native code (compiled at `-O3`) and 33 times faster than Numba.

Table 6.21: LINPACK benchmark mean (M) MFLOPS on Intel i5 (macOS) for Python (CPython), Cython, Numba, Olympus and native C, with standard deviation (SD)

Technology	MFLOPS (M)	SD
Python (CPython)	12.22	0.5943
Cython (-O3)	20.58	0.2598
Numba (JIT)	81.33	4.592
Olympus (-O3)	2663	118.6
Native C (-O3)	3047	147.5

Whilst it should be possible to increase the runtime performance of the Cython binary by using *type hints* and `ctypes`, the Olympus code does not require the programmer to add this information to achieve this level of runtime performance. The key point is that although Olympus was designed for micro-core architectures, it generates binaries that are significantly faster than Cython and Numba on the desktop.

These results confirm that the Olympus design can not only scale from 32-bit micro-cores to 64-bit desktop CPUs but also out-perform comparable Python compilation technologies, both in terms of binary code size and runtime performance, for the LINPACK benchmark.

6.7 Summary

The Olympus abstract machine delivers kernel runtimes that can, on average, match native C for the Fibonacci and LINPACK benchmarks ($GM = 101.0\%$) across eight different CPU architectures (Section 6.5.2). Furthermore, this is achieved whilst generating kernel binaries that are, on average, between 1.4 times ($GM = 1.424$) and two times ($GM = 2.028$) bigger than native C across the CPU architectures (Section 6.5.1). These results are compelling; the Olympus abstract machine closes the performance gap of ePython codes with native C, whilst at the same time as delivering binary code sizes that not only fit within the micro-core’s on-chip RAM, for standard benchmarks such as LINPACK, but also are able to benefit from the Olympus dynamic code loading approach (Section 6.4.1). Furthermore, these results were obtained from a single code for each benchmark running on all of the different technologies listed in Section 5.3.1. Additionally, ePython can now deliver this level of runtime performance whilst providing the programmer with significantly increased productivity imparted by the dynamic code loading and memory hierarchy support to manage arbitrarily-sized code and data on idiosyncratic micro-core devices, in comparison to manually managing these techniques in native C using different, CPU-specific APIs and libraries. These capabilities significantly increase the dynamic language applications that can be deployed to micro-cores. For example, the ability to run the ML cancer detection codes on full-sized images.

The challenge with evaluation against other prior work beyond C is that such technologies do not support the tiny memory spaces of the target micro-core devices. Comparable technologies discussed in Chapter 2, such as Numba and MicroPython, require far more memory, as discussed in Section 6.6.1. For example, MicroPython requires at least 256KB [99], eight times more memory than the target devices of this work. However, over and above the comparison of Olympus to the comparable desktop technologies discussed in Section 6.6, some general comparisons can be drawn based on previously published work; for instance, the MicroPython versus C runtime performance comparison [96] reveals that MicroPython is approximately 87 times slower than C. Assuming that these values are for the interpreter, and that the *Viper* code generator is 7 times faster [126], MicroPython is still 12 times slower than hand-crafted C. In comparison, Olympus matches native C runtime performance for the Fibonacci and LINPACK benchmarks across the CPU architectures (Figure 6.17), and, as a worst case, is only around 4 times slower than native C on the RISCv64 for the Sieve bench-

mark (Table 6.18). In comparison to Micropython’s *Viper* compiler, this is a significantly higher level of runtime performance. Furthermore, when we consider that the Olympus dynamic code loading approach can deploy kernels that have a smaller initial footprint than the equivalent native C binaries (Section 6.4.1), the approach is even more compelling in comparison to the minimum 256KB code and 8KB RAM (for the stack) required by MicroPython [99].

The [micro-core] architecture presents many features and constraints which contribute to software design challenges for the application developer. Addressing these challenges within the software stack that supports application development is critical to improving productivity and expanding the range of applications for the architecture.

Ritchie and Ross, “Advances in Run-Time Performance and Interoperability for the Adapteva Epiphany Coprocessor” [108]

7

Conclusions and future work

7.1 Conclusions

This thesis has explored the investigation, design and implementation of new programming techniques that simplify the development of codes on micro-core architectures using dynamic programming languages, such as Python, whilst closing the associated runtime performance gap with codes written natively in C. Specifically, this thesis addressed the following research questions regarding the design and implementation of dynamic languages on micro-cores:

RQ1 How to manage the limited on-chip memory for data?

RQ2 How to manage the limited on-chip memory for code?

RQ3 How to address the low runtime performance of virtual machines?

RQ4 How to manage the idiosyncratic architectures of micro-core architectures?

This is driven by micro-core architectures providing an interesting option for overcoming the issues imposed by the end of Moore's Law [186] and Dennard Scaling [187]. However, if the first two research questions are not addressed, these architectures remain untenable; without the ability to overcome the memory limitations of micro-cores, their application to solving real-world problems using dynamic languages is extremely limited, as demonstrated by the limited use of the Adapteva Epiphany-III. This field is moving very fast and since this research began, RISC-V has grown significantly and the Epiphany is becoming obsolete. Initially, it appeared that RISC-V would play a role in the embedded domain, rather than HPC, but now with the development of the European Processor Initiative (EPI) RISC-V accelerator [188], its role is becoming more prominent in HPC.

As demonstrated in Section 6.2, the support of memory hierarchies within ePython was critical in enabling a cancer cell detection code to operate on full-size (30MB) images, rather than being limited to smaller, 3600 pixel, interpolated images. Furthermore, the ability to deploy codes of arbitrary size using the dynamic code loading support within the Olympus abstract machine reduced the initial memory footprint of kernels by up to 30% over those developed using native C, thereby freeing up more precious on-chip memory for data (demonstrated by the Jacobi benchmark results in Section 6.4).

Whilst the ePython virtual machine provided a productive environment to deploy parallel codes written in a dynamic language to micro-core architectures, the runtime performance overhead of the interpreter limited its use for real-world codes. There are several potential approaches to increasing the runtime performance of dynamic languages, and this research began with the question of AOT or JIT compilation. There is prior work here, for example MicroPython performs the compilation of bytecode to native code on the device [126] similar to JIT except that the bytecode is not profiled as is common for JIT compilers, rather the bytecode is just lowered to native code. However, an alternative approach was taken for this work, similar to that employed by the Pallene / Titan compiler [138] for Lua [74]. Here, the source language compiler, running on the host, emits C source code that is then compiled to generate native binary executables. For ePython, this approach resulted in kernel runtime performance that was comparable to or, in some cases could exceed, native C kernels, as confirmed for the LINPACK benchmark in Section 6.3.2.6, and, at a worst-case, was around four times slower than native C for the Sieve of Eratosthenes benchmark, at compiler optimisation level $-O3$ (Section 6.3.2.8). However, one should bear in mind that developing kernels in native C requires an extensive understanding of C memory structures, linker scripts and the associated device SDKs. Furthermore, a single Python code is portable across these architectures, which is not the case for the standard C codes. Crucially, as demonstrated in Section 6.3.1, the Olympus abstract machine delivered runtime performance, on average (*GM*), of around 75% that of native C across four benchmarks on eight CPU architectures (Section 6.5.2) and over 300 times that of the ePython virtual machine for an iterative Fibonacci code, with an overall memory footprint up to 17 times smaller than the VM (Section 6.3.1).

This work has also addressed the final concern, namely the portability of user codes and underlying runtime support. A mix of 32-bit, 64-bit, little-endian, big-endian, CISC and RISC CPU architectures were specifically chosen to run the benchmarks to exercise portability issues for the Olympus abstract machine. All of the benchmarks run unmodified across all the supported platforms and the Olympus abstract machine builds from a single codebase, which results in significant programmer productivity gains. All device-specific code is managed within the mnemonics and runtime support functions, with the generated Olympus abstract machine code remaining the same across all platforms. Furthermore, the ePython virtual machine was also shown to be portable to several of micro-core architectures (Adapteva Epiphany-III, Xilinx MicroBlaze and PicoRV32 RISC-V) with the minimum of effort.

Consequently it is concluded that, based on this research, one is able to *have their cake and eat it*. Namely, it is possible for programmers to develop codes in productive dynamic languages, such as Python, and for these to run without modification across many micro-core architectures, resulting in runtime performance similar to native code, whilst utilising unlimited code and data sizes.

7.2 Summary of contributions

The extremely limited amount of on-chip memory (between 32KB and 64KB) presents significant challenges to implementing dynamic languages on micro-core architectures. Overcoming these challenges requires a holistic approach to design and implementation for both the host and device components. As much heavy lifting as possible needs to be performed on the host, with the abstract machine / runtime being as simple as possible to reduce code size and increase runtime performance as much as possible. Whilst there are a number of approaches to accelerating dynamic languages on embedded / IoT devices, including MicroPython [85], they are still too large (c. 256KB for MicroPython) to deploy to micro-core devices, such as the Adapteva Parallella. The key contributions of this work for the design and implementation of dynamic languages on micro-core architectures are:

- The first *pass by reference* semantics model for kernels offloaded to micro-core accelerators, addressing RQ1.
- Support for arbitrary-sized data through a compact model for device memory hierarchy support, addressing RQ1.
- Support for arbitrary-sized kernel codes through a dynamic loading model for natively generated functions, addressing RQ2.
- The Olympus portable, compact (on average between 1.4 and 2 times larger than native C), high-performance abstract machine (on average (*GM*), 75% of the runtime performance of native C) and an associated memory addressing model for dynamic programming languages, addressing RQ3.
- The Eithne micro-core benchmarking framework, addressing RQ4.

7.3 Research impact

In addition to the contributions of this work, the technologies developed as part of the project have also had wider impact. The Olympus ePython native code compiler and abstract machine have formed part of *Vipera* [189], a framework for implementing dynamic languages on micro-core architectures. *Vipera* is a deliverable of the *ePython commercialisation* project, funded by EPSRC (Engineering and Physical Sciences Research Council) via an *Impact Acceleration Account (IAA)*, to widen the developer community and provide the basis for future commercialisation and industry collaboration for vPython¹ and related Olympus technologies.

The Eithne [190] benchmarking framework and the Cerberus [31] FPGA RISC-V micro-core design outputs of this research project will form part of the *RISC-V testbed* project for the *ExCALIBUR Hardware and Enabling Software (H&ES) Programme* [2]. Eithne will provide the framework for users to deploy and run benchmark codes on the soft-core, multi-core RISC-V processors, based on the Cerberus design, deployed on the FPGA cards within the testbed. Coupled with *Vipera*, these technologies will significantly reduce the effort required to develop and deploy codes in Python, C and Fortran to soft-core and physical RISC-V processors, supporting the HPC developer community in the drive towards *exascale* computing.

7.4 Limitations

7.4.1 ePython language

ePython is a subset of Python 2.7, with some version 3 features such as `nonlocal` variable support. However, a major limitation in terms of Python compatibility is the lack of support for Object-Oriented Programming. Whilst the Olympus abstract machine can support a simple OO model (Section 4.3.3.5), adding full Python OO programming features to ePython would require support for dynamic method dispatch within the Olympus abstract machine, coupled with additional functionality in the ePython compiler to manage classes and their associated inheritance hierarchies.

¹The development of ePython to leverage the features of *Vipera*, including seamless C integration and separately compiled module support.

7.4.2 Type inferencing

The type inferencing model for ePython is simple due to its restricted type system; ePython does not support classes, polymorphic composite types (e.g. Python lists containing different element types) or polymorphic function return types (multiple types returned from a function). Thus, functions can only have a single return type or return values that can be coerced to a single type. A more complex type inferencing model [191] would be required to support these features, coupled with the inclusion of dynamic fallback [112] with boxed types, particularly for polymorphic lists. Furthermore, whilst Olympus supports dynamically typed programming languages, they must be *strongly typed* per Python, Perl or Ruby, rather than *weakly typed* per JavaScript.

7.4.3 C integration / Olympus application binary interface

Whilst integration with C is provided by ePython, the Olympus abstract machine defines its own application binary interface (ABI) for the generated functions². Although this allows closure support across C compilers, the consequence is that Olympus function calls are incompatible with the native C ABI. This precludes generating object files that are directly compatible with C codes and programmers need to be aware of Olympus ABI (memory model, including environments / function stack) to call Olympus functions from C codes.

7.4.4 Limited micro-core architecture evaluation

The Cerberus micro-core design was developed (Section 5.2.1) due to the lack of additional micro-core architectures, over and above the Adapteva Epiphany-III, available at the beginning of this research. The Xilinx MicroBlaze and RISC-V (PicoRV32) based versions of Cerberus provided flexible development and testing platforms for the Olympus abstract machine, particularly with respect to the portability of the technology across architectures. However, coupled with the Epiphany-III, the Cerberus designs only provided a limited set of micro-core architectures for the evaluation of the Olympus abstract machine.

²Function parameters and return values are managed by Olympus within the environment frames, rather than by the C compiler directly.

7.5 Future work

7.5.1 Automatic memory management for data and code

Whilst this work demonstrated the importance of prefetching, further exploration is advised. This is not an easy topic [106], and likely automatic approaches to do this will be highly architecture specific. Furthermore, the dynamic loading and unloading of kernel functions is manually performed by the programmer. Ideally, the compiler should analyse the data access patterns and automatically select the optimal memory transfer profile, or *tiling* [106] and [192], for each kernel. Correspondingly, writing efficient parallel codes is difficult and techniques that select an optimal code implementation strategy [193], coupled with the automatic management of the dynamic loading [194] and unloading [195] of kernel functions, would not only simplify the development of codes on micro-cores but would also maximise the amount of precious on-chip memory available for data on these architectures.

7.5.2 Optimisation of the Olympus abstract machine

7.5.2.1 Mnemonics: size versus runtime performance

Whilst the Olympus abstract machine has been proven to provide a high level of runtime performance on micro-core architectures, the generated binary code is up to 4 times larger than the same kernel coded in native C (Section 6.5.1). As kernel functions can be loaded and unloaded dynamically this is not a major blocker, however it would be beneficial to explore this aspect in more detail and optimise further. Although there is the trade-off between code size and runtime performance, it would be beneficial to provide programmers with a more granular option to select code size over runtime performance over and above the `-Os` and `-O3` C compiler optimisation levels. This could be implemented globally, at a file level or for each function. At present, the Olympus mnemonics are implemented as a mixture of C macros and functions, with the majority being the former. Whilst this delivers the required runtime performance, the mnemonics produce *wordy* C source code that inflates the binary code size. An approach would be to implement all of the mnemonics as C functions that can be inlined or called via standard function call, as selected by the programmer. The former option would provide the same runtime performance as the current macro-based model, with the latter providing the ability to deliver smaller kernel binaries. This approach could be combined with the automated selection of the C compiler's optimisation flags

similar to that performed by AcovEA [196] and AcovSA [197], which optimise the flags over a number of execution runs to provide the ideal combination that maximise runtime performance. Furthermore, the optimiser could be configured to balance runtime performance and code size, further enhancing the applicability of this technique to native code generation for micro-core architectures.

7.5.2.2 Automatic dynamic function selection

The Olympus dynamic function loading support allows the programmer to overcome the limit of kernel code size imposed by micro-core device memory. However, to fully optimise the memory available for data, the programmer must manually implement the dynamic loading and freeing strategy. The compiler-assisted, adaptive runtime support approach of [51] provides an average 27% reduction in the deployed kernel size by only linking the OpenMP support functions specifically required by the kernel. Whilst this adaptive runtime support function mechanism is designed for statically-linked OpenMP kernels, a similar method of kernel analysis could be applied for Olympus dynamically loaded functions. Furthermore, as the ePython host-based function loader parses ELF files to extract the required dynamic function requested by the device, there is no additional development required to separate out runtime library functions into separate object files to support adaptive runtime support. Moreover, as the compiler already adds external function references to the `<filename>_oly_externs.h` file (outlined in Section 4.3.3.6), the current function reference management could be updated to insert dynamic function loading requests or static references based on heuristic selection criteria.

7.5.2.3 Data types

On micro-core and embedded devices, the design of data structures can have a major impact on the number of elements that can be supported. The Sieve of Eratosthenes benchmark (Section 5.3.3.5) highlighted, on the Epiphany-III, that the native C benchmark can support the full 8190 flag array size as it uses byte (`char`) elements, whereas ePython / Olympus use `int` (4 byte) elements, requiring the full 32KB of the on-chip memory. The ability to specify other data types other than `integer` or `real` and `complex` for arrays, such as `byte`, from within ePython would allow the programmer to deploy kernels with size-optimised data structures. There are no intrinsic limitations within the Olympus abstract machine to providing this support, as it already supports

`char` arrays for strings and has been designed to support size-specific data types, such as `uint8_t` and `int16_t`. The only modifications required are the definition of new mnemonics for the required type and arrays of the type. The source language (ePython) would need to be updated to provide a mechanism to specify the type. In the specific case of byte arrays, a standard function that returns a new byte array and updates to the compiler type management would suffice³, as the new mnemonics would allow access to arrays and array elements of the new type.

7.5.2.4 Register allocation

At present, all CPU register allocation is performed by the C compiler. Therefore, the runtime performance impact of manual register allocation / hinting for critical Olympus abstract machine variables (current frame / stack pointer, return stack and heap pointer) is unknown. Although the current model simplifies the code generation rules within the compiler, it would be valuable to confirm if this is optimal for CPU architectures with a limited register set (e.g. x86) particularly for the `-Os` C compiler optimisation level. In this manner it might be possible to increase runtime performance further for specific codes.

7.5.3 Compiler intermediate representation

Currently, a number of languages, including Julia [198] and Haskell [139], generate LLVM intermediate representation (LLVM IR) code [94], which is then compiled by LLVM to provide a native binary. The decision to generate C source code rather than LLVM IR for the Olympus abstract machine was driven by the fact that LLVM does not support two of the target micro-core architectures (Adapteva Epiphany-III and Xilinx MicroBlaze). Whilst Olympus supports a number of C compilers over and above GCC, including Clang, the fact that the Glasgow Haskell Compiler (GHC) LLVM code generator can out-perform the previous C-based generator by nearly 3 times [137] suggests that it would be beneficial to understand the trade-off between the simplicity of generating abstract machine mnemonics in C and the runtime performance benefits of directly generating LLVM IR code, per GHC. More widely, the Olympus abstract machine could utilise Multi-Level Intermediate Representation (MLIR) [199], which will generate both LLVM-IR and C code (EmitC dialect [200]). Lowering through MLIR

³The AST needs to be aware of byte and arrays of bytes to support the correct type inferencing and the generation of element access code.

dialects could help encode aspects such as register allocation in a more bespoke manner but one in which the infrastructure is shared with other MLIR technologies and approaches.

7.5.4 Additional device support: GPUs and FPGAs

One advantage of generating C source code is the ability to target C-based languages, such as OpenCL C [47] and Xilinx HLS C [64], thereby allowing Olympus to target GPUs and FPGAs. However, targeting OpenCL C and HLS C isn't a trivial update to the existing C abstract machine code generator as both languages have restricted C language support that impacts the underlying Olympus mnemonic implementation. For example, neither language supports function pointers, which are currently used within the Olympus abstract machine. OpenCL C++ does support limited function pointers and lambdas but doesn't support kernel functions calling other kernel functions [201]. Therefore, the Olympus function dispatch model would require modification to use C *labels* and *goto* jumps. However, the current function dispatch model was designed with this restriction in mind and was one of the deciding factors in managing the function stacks, parameters and return values within the Olympus abstract machine environment, rather than relying on the underlying C function ABI management.

Bibliography

- [1] Tackling the power and energy wall for future HPC systems. [Online]. Available: <https://www.hpcwire.com/2013/12/17/tackling-power-energy-wall-future-hpc-systems/>
- [2] ExCALIBUR - Exascale Computing ALgorithms & Infrastructures Benefiting UK Research. [Online]. Available: <https://excalibur.ac.uk/>
- [3] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, “Scaling the power wall: A path to exascale,” in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841, ISSN: 2167-4337.
- [4] Frontier to meet 20mw exascale power target set by DARPA in 2008. [Online]. Available: <https://www.hpcwire.com/2021/07/14/frontier-to-meet-20mw-exascale-power-target-set-by-darpa-in-2008/>
- [5] B. D. de Dinechin, “Kalray MPPA®: Massively parallel processor array: Revisiting DSP acceleration with the kalray MPPA manycore processor,” in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–27.
- [6] X. Zhu, Y. Zhao, and P. Balaji, “Probing the underlying implementation mechanisms of SW26010,” in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 694–699.
- [7] A. Olofsson, “Epiphany-v: A 1024 processor 64-bit RISC system-on-chip.” [Online]. Available: <http://arxiv.org/abs/1610.01832>

- [8] PEZY-SC2 - PEZY - WikiChip. [Online]. Available: <https://en.wikichip.org/wiki/pezy/pezy-scx/pezy-sc2>
- [9] June 2021 | TOP500. [Online]. Available: <https://www.top500.org/lists/green500/2021/06/>
- [10] (2019) June 2019 — GREEN500 Supercomputer Sites. [Online]. Available: <https://www.top500.org/green500/lists/2019/06/>
- [11] (2019) June 2019 — TOP500 Supercomputer Sites. [Online]. Available: <https://www.top500.org/lists/2019/06/>
- [12] A. Joglekar, G. Gurralla, P. Kumar, F. C. Joseph, T. S. Kiran, K. R. Sahasranand, and H. Tyagi, “Open-source heterogeneous constrained edge-computing platform for smart grid measurements,” vol. 70, pp. 1–12, conference Name: IEEE Transactions on Instrumentation and Measurement.
- [13] Adapteva, “E16g301 epiphany 16-core microprocessor,” http://www.adapteva.com/docs/e16g301_datasheet.pdf, 2013, accessed: 2018-07-25.
- [14] “xCORE™ : Architecture overview,” 2013.
- [15] P. Brauer, M. Lundqvist, and A. Mällo, “Improving latency in a signal processing system on the epiphany architecture,” in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 796–800, ISSN: 2377-5750.
- [16] M. Kudlur and S. Mahlke, “Orchestrating the execution of stream programs on multicore platforms,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’08. ACM, pp. 114–124. [Online]. Available: <http://doi.acm.org/10.1145/1375581.1375596>
- [17] S. Lyberis, P. Pratikakis, D. S. Nikolopoulos, M. Schulz, T. Gamblin, and B. R. de Supinski, “The myrmics memory allocator: Hierarchical, message-passing allocation for global address spaces,” in *Proceedings of the 2012 International Symposium on Memory Management*, ser. ISMM ’12. ACM, 2012, pp. 15–24. [Online]. Available: <http://doi.acm.org/10.1145/2258996.2259001>

- [18] L. I. Xii, R. Banakar, R. Banakar, S. Steinke, S. Steinke, B.-s. Lee, B.-s. Lee, M. Balakrishnan, M. Balakrishnan, P. Marwedel, and P. Marwedel, "Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area and Energy Consumption," 2001.
- [19] N. Brown, "ePython: An Implementation of Python for the Many-Core Epiphany Co-processor," in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, Nov. 2016, pp. 59–66.
- [20] M. Jamieson, N. Brown, and S. Liu, "Having your cake and eating it: Exploiting Python for programmer productivity and performance on micro-core architectures using ePython," in *Proceedings of the 19th Python in Science Conference : SciPy 2020*, 2020, pp. 97–105. [Online]. Available: <https://www.research.ed.ac.uk/en/publications/having-your-cake-and-eating-it-exploiting-python-for-programmer-p>
- [21] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, Aug. 2003. [Online]. Available: <http://doi.wiley.com/10.1002/cpe.728>
- [22] M. Jamieson and N. Brown, "High level programming abstractions for leveraging hierarchical memories with micro-core architectures," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 128–138, 2020. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0743731518305616>
- [23] A. Ranta, *Implementing Programming Languages. an Introduction to Compilers and Interpreters*. College Publications, 2012.
- [24] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies." Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Tech. Rep. SAND2015-1862C, 2015. [Online]. Available: <https://www.osti.gov/biblio/1245908>
- [25] M. Jamieson. Eithne micro-core benchmarking framework. [Online]. Available: <https://gitlab.com/mjamieson/eithne>

- [26] M. Jamieson and N. Brown. (2022) Performance of the Vipera framework for DSLs on micro-core architectures. This preprint to the DSL-HPC workshop has not undergone any post-submission improvements or corrections. [Online]. Available: <https://arxiv.org/abs/2209.00894>
- [27] —, “Compact native code generation for dynamic languages on micro-core architectures,” in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. Association for Computing Machinery, 2021, pp. 131–140. [Online]. Available: <https://doi.org/10.1145/3446804.3446853>
- [28] —, “Benchmarking micro-core architectures for detecting disasters at the edge,” in *2020 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, 2020, pp. 27–35.
- [29] —, “Poster 99: Eithne: A framework for benchmarking Micro-Core Accelerators,” *Supercomputing 2019 (SC19)*, Nov 2019. [Online]. Available: https://sc19.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost186.html
- [30] N. Brown and M. Jamieson, “Leveraging hierarchical memories for micro-core architectures,” *5th International Conference on Exascale Applications and Software*, 2018.
- [31] M. Jamieson. Cerberus RISC-V micro-core. [Online]. Available: <https://gitlab.com/mjamieson/cerberus.git>
- [32] C. Mutigwe, J. Kinyua, and F. Aghdasi, “Instruction Set Usage Analysis for Application-Specific Systems Design,” vol. 7, no. 2, p. 5, 2013.
- [33] M. Slater, Ed., *A Guide To RISC Microprocessors*, 2nd ed. Academic Press, 1992, ch. 1, p. 7.
- [34] Standard performance evaluation corporation SPEC CPU2006. [Online]. Available: <https://www.spec.org/cpu2006/src.alt/>
- [35] O. Lempel, A. Peleg, and U. Weiser, “Intel’s MMX/sup TM/ technology-a new instruction set extension,” in *Proceedings IEEE COMPCON 97. Digest of Papers*, pp. 255–259, ISSN: 1063-6390.

- [36] J. Tyler, J. Lent, A. Mather, and H. Nguyen, “AltiVec/sup TM/: bringing vector technology to the PowerPC/sup TM/ processor family,” in *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*, pp. 437–444, ISSN: 1097-2641.
- [37] MIPS64 architecture – MIPS. [Online]. Available: <https://www.mips.com/products/architectures/mips64/>
- [38] M. Livesey, F. Costen, and Xiaoling Yang, “Performance of streaming SIMD extensions instructions for the FDTD computation,” vol. 54, no. 3, pp. 160–168. [Online]. Available: <http://ieeexplore.ieee.org/document/6293973/>
- [39] A. Strey and M. Bange, “Performance analysis of intel’s MMX and SSE: A case study,” in *Euro-Par 2001 Parallel Processing*, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Springer Berlin Heidelberg, vol. 2150, pp. 142–147, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-44681-8_21
- [40] Intel®; instruction set extensions technology. [Online]. Available: <https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html>
- [41] M. Alvanos and P. Trancoso, “Video SIMDBench: Benchmarking the compiler vectorization for multimedia applications,” in *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 168–175.
- [42] M. Boettcher, B. M. Al-Hashimi, M. Eyole, G. Gabrielli, and A. Reid, “Advanced SIMD: Extending the reach of contemporary SIMD architectures,” in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–4, ISSN: 1558-1101.
- [43] (2014, Mar.) Epiphany Architecture Reference. [Online]. Available: http://www.adapteva.com/docs/epiphany_arch_ref.pdf
- [44] 20+ GFLOPS/watt: More processing power for less watts. [Online]. Available: <https://streamhpc.com/blog/2012-08-27/processors-that-can-do-20-gflops-watt/>
- [45] K. L. Labowski, P. W. Jungwirth, J. A. Ross, and D. A. Richie, “Implementing hilbert transform for digital signal processing on epiphany many-core coproces-

- sor,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–6.
- [46] J. Walton. (2021) Graphics card power consumption and efficiency tested. [Online]. Available: <https://www.tomshardware.com/uk/features/graphics-card-power-consumption-tested>
- [47] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” vol. 12, no. 3, 2010, conference Name: Computing in Science Engineering.
- [48] (2013) Openmp application program interface version 4.0. <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. OpenMP Architecture Review Board.
- [49] D. Richie and J. Ross, “OpenCL + OpenSHMEM hybrid programming model for the adapteva epiphany architecture,” vol. 10007, pp. 181–192. [Online]. Available: <http://arxiv.org/abs/1608.03549>
- [50] S. N. Agathos and V. V. Dimakopoulos, “Adaptive OpenMP runtime system for embedded multicores,” in *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 174–181.
- [51] S. N. Agathos, V. V. Dimakopoulos, and I. K. Kasmeridis, “Compiler-assisted, adaptive runtime system for the support of OpenMP in embedded multicores,” vol. 110, p. 102895. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819122000035>
- [52] (2020, Aug.) parallella/otp. Original-date: 2016-10-05T02:56:10Z. [Online]. Available: <https://github.com/parallella/otp>
- [53] A. E. Ritz. (2020, Aug.) drewvid/parallella-lisp. Original-date: 2015-12-29T23:42:15Z. [Online]. Available: <https://github.com/drewvid/parallella-lisp>
- [54] *The OpenACC Application Programming Interface version 2.5*, 2015. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf
- [55] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy. (2020) NVIDIA ampere architecture in-depth. [Online]. Available: <https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>

- [56] N. Sakharnykh. (2017) Maximizing unified memory performance in CUDA. [Online]. Available: <https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/>
- [57] Famous graphics chips: Nvidia's GeForce 256 | IEEE computer society. [Online]. Available: <https://www.computer.org/publications/tech-news/chasing-pixels/nvidias-geforce-256/>
- [58] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," vol. 38, no. 8, pp. 391–407, 2012. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819111001335>
- [59] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," vol. 28, no. 2, pp. 39–55, 2008, conference Name: IEEE Micro.
- [60] (2017) CUDA zone. [Online]. Available: <https://developer.nvidia.com/cuda-zone>
- [61] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.
- [62] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 494–506, ISSN: 2378-203X.
- [63] S. Akthar. FPGA architecture. [Online]. Available: <https://allaboutfpga.com/fpga-architecture/>
- [64] "Vivado design suite user guide: High-Level synthesis," p. 589. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf
- [65] The OpenCL™ C specification. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html

- [66] SYCL™ 2020 specification (revision 4). [Online]. Available: <https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [67] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, “Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis,” vol. 5, pp. 2747–2762, 2017, conference Name: IEEE Access.
- [68] “PERC: Posit enhanced rocket chip,” p. 8.
- [69] J. Gray, “GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator,” in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium On.* IEEE, 2016, pp. 17–20.
- [70] Welcome to Python.org. [Online]. Available: <https://www.python.org/>
- [71] R: What is r? [Online]. Available: <https://www.r-project.org/about.html>
- [72] Ruby programming language. [Online]. Available: <https://www.ruby-lang.org/en/>
- [73] (2021) ECMA-262. [Online]. Available: <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
- [74] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “The implementation of lua 5.0,” p. 18. [Online]. Available: <https://www.lua.org/doc/jucs05.pdf>
- [75] index | TIOBE - the software quality company. [Online]. Available: <https://www.tiobe.com/tiobe-index/>
- [76] L. Prechelt, “An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program,” Fakultät für Informatik, Universität Karlsruhe, Tech. Rep. Technical Report 2000-5, Mar. 2000. [Online]. Available: <http://page.mi.fu-berlin.de/prechelt/Biblio/jccpprtTR.pdf>
- [77] D. A. Richie, “COPRTHR® API reference,” 2014.
- [78] J.-W. Buurlage, T. Bannink, and A. Wits, “Bulk-synchronous pseudo-streaming algorithms for many-core accelerators,” 2016. [Online]. Available: <http://arxiv.org/abs/1608.07200>

- [79] S. N. Agathos, A. Papadogiannakis, and V. V. Dimakopoulos, “Targeting the parallella,” in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Springer Berlin Heidelberg, 2015, vol. 9233, pp. 662–674. [Online]. Available: http://link.springer.com/10.1007/978-3-662-48096-0_51
- [80] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, “Parallel programming model for the epiphany many-core coprocessor using threaded MPI,” in *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, ser. MES '15. ACM, 2015, pp. 41–47. [Online]. Available: <http://doi.acm.org/10.1145/2768177.2768183>
- [81] V. M. Ionescu and F. M. Enescu, “Investigating the performance of MicroPython and c on ESP32 and STM32 microcontrollers,” in *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp. 234–237, ISSN: 2642-7036.
- [82] M. Bysiek, A. Drozd, and S. Matsuoka, “Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints,” in *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pp. 9–18.
- [83] D. Beazley, “Understanding the python GIL,” 2010, PyCon 2010. [Online]. Available: <http://www.dabeaz.com>
- [84] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The Best of Both Worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, Mar. 2011. [Online]. Available: <http://ieeexplore.ieee.org/document/5582062/>
- [85] (2018) MicroPython - Python for microcontrollers. [Online]. Available: <http://micropython.org/>
- [86] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.
- [87] B. Catanzaro, M. Garland, and K. Keutzer, “Copperhead: compiling an embedded data parallel language,” vol. 46, no. 8, pp. 47–56. [Online]. Available: <https://doi.org/10.1145/2038037.1941562>

- [88] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha, “Parakeet: a just-in-time parallel accelerator for python,” in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, ser. HotPar’12. USENIX Association, p. 14.
- [89] D. Jacob, P. Trinder, and J. Singer, “Python programmers have GPUs too: automatic python loop parallelization with staged dependence analysis,” in *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2019. Association for Computing Machinery, pp. 42–54. [Online]. Available: <https://doi.org/10.1145/3359619.3359743>
- [90] pycuda 2021.1 documentation. [Online]. Available: <https://documen.tician.de/pycuda/>
- [91] G. Seif. (2019, Jun.) Here’s how you can get some free speed on your Python code with Numba. [Online]. Available: <https://bit.ly/3Cq5DB0>
- [92] A ~5 minute guide to Numba — Numba 0.50.1 documentation. [Online]. Available: <https://numba.pydata.org/numba-doc/latest/user/5minguide.html>
- [93] CodePy. [Online]. Available: <https://mathema.tician.de/software/codepy/>
- [94] The LLVM Compiler Infrastructure Project. [Online]. Available: <https://llvm.org/>
- [95] Parallel thread execution ISA version 7.6. Archive Location: Programming Guides. [Online]. Available: <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>
- [96] micropython/micropython. [Online]. Available: <https://github.com/micropython/micropython>
- [97] D. George. Update 4: The 3 different code emitters · Micro Python: Python for microcontrollers. [Online]. Available: <https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers/posts/664832>
- [98] ——. Update 5: The 3 different code emitters, part 2 · Micro Python: Python for microcontrollers. [Online]. Available: <https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers/posts/665145>

- [99] FAQ · micropython/micropython wiki. [Online]. Available: <https://github.com/micropython/micropython>
- [100] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, “Optimistic parallelism requires abstractions,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, pp. 211–222, event-place: San Diego, California, USA. [Online]. Available: <http://doi.acm.org/10.1145/1250734.1250759>
- [101] N. Brown, “Offloading python kernels to micro-core architectures,” supercomputing 2017. [Online]. Available: https://sc17.supercomputing.org/SC17%20Archive/tech_poster/poster_files/post180s2-file3.pdf
- [102] ——. (2017) Offloading kernels from python codes onto the epiphany. [Online]. Available: <https://www.parallella.org/2017/09/29/offloading-kernels-from-python-codes-onto-the-epiphany/>
- [103] ——. (2017) Epiphany as an accelerator: managing device data. Original-date: 2015-07-08T14:42:58Z. [Online]. Available: <https://github.com/mesham/epython>
- [104] (2013) Unified memory in CUDA 6. [Online]. Available: <https://developer.nvidia.com/blog/unified-memory-in-cuda-6>
- [105] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic management of scratch-pad memory space,” in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. ACM, pp. 690–695. [Online]. Available: <http://doi.acm.org/10.1145/378239.379049>
- [106] L. Li, H. Wu, H. Feng, and J. Xue, “Towards data tiling for whole programs in scratchpad memory allocation,” in *Advances in Computer Systems Architecture*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 63–74. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-74309-5_8
- [107] T. Vocke, “AN EVALUATION OF THE ADAPTEVA EPIPHANY MANY-CORE ARCHITECTURE,” Master’s thesis, 2015. [Online]. Available: https://essay.utwente.nl/68024/1/vocke_MA_EEMCS.pdf

- [108] D. A. Richie and J. A. Ross, “Advances in run-time performance and interoperability for the adapteva epiphany coprocessor,” vol. 80, pp. 1531–1541. [Online]. Available: <http://arxiv.org/abs/1604.04207>
- [109] Numba and types — numba 0.12.2 documentation. [Online]. Available: https://numba.pydata.org/numba-doc/0.12.2/tutorial_types.html
- [110] V. Chen. (2020, Oct.) vinta/awesome-python. Original-date: 2014-06-27T21:00:06Z. [Online]. Available: <https://github.com/vinta/awesome-python>
- [111] X. Xia, X. He, Y. Yan, L. Xu, and B. Xu, “An Empirical Study of Dynamic Types for Python Projects,” in *Software Analysis, Testing, and Evolution*, ser. Lecture Notes in Computer Science, L. Bu and Y. Xiong, Eds. Cham: Springer International Publishing, 2018, pp. 85–100.
- [112] A. Rastogi, A. Chaudhuri, and B. Hosmer, “The ins and outs of gradual type inference,” vol. 47, no. 1, pp. 481–494, 2012. [Online]. Available: <https://doi.org/10.1145/2103621.2103714>
- [113] L. Cheng, B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten, “Type freezing: exploiting attribute type monomorphism in tracing JIT compilers,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, Feb. 2020, pp. 16–29. [Online]. Available: <https://doi.org/10.1145/3368826.3377907>
- [114] M. Salib, “Faster than c: Static type inference with starkiller,” in *PyCon Proceedings, Washington DC*. SpringerVerlag, 2004, pp. 2–26.
- [115] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, “Type inference for static compilation of JavaScript,” vol. 51, no. 10, pp. 410–429, 2016. [Online]. Available: <https://doi.org/10.1145/3022671.2984017>
- [116] S. L. Peyton Jones and J. Launchbury, “Unboxed values as first class citizens,” in *ACM Conference on Functional Programming and Computer Architecture (FPCA’91)*, ser. Lecture Notes in Computer Science, vol. 523. Springer, 1991, pp. 636–666.

- [117] L. Cardelli, “Basic polymorphic typechecking,” vol. 8, no. 2, pp. 147–172, 1987. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0167642387900190>
- [118] S. L. Peyton Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., 1987, pp. 171–182.
- [119] I. JTC1/SC22/WG14, *Programming Languages—C*, issue: SO/IEC 9899:TC3. [Online]. Available: <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf>
- [120] P. K. Sinha, *Distributed Operating Systems: Concepts and Design*, 1st ed. Prentice-Hall of India Pvt.Ltd, 2009, ch. 3, pp. 122–125.
- [121] ———, *Distributed Operating Systems: Concepts and Design*, 1st ed. Prentice-Hall of India Pvt.Ltd, 2009, ch. 8, pp. 410–414.
- [122] (2013) Epiphany SDK (ESDK). [Online]. Available: <https://www.adapteva.com/epiphany-sdk/>
- [123] Documentation/contiguous-memory.txt - kernel/exynos - git at google. [Online]. Available: <https://android.googlesource.com/kernel/exynos/+android-exynos-3.4/Documentation/contiguous-memory.txt>
- [124] Python/c API reference manual — python 2.7.2 documentation. [Online]. Available: <https://python.readthedocs.io/en/v2.7.2/c-api/index.html>
- [125] (2019) PYNQ-Z2. [Online]. Available: <http://www.tul.com.tw/ProductsPYNQ-Z2.html>
- [126] (2013) Update 4: The 3 different code emitters · micro Python: Python for microcontrollers. [Online]. Available: <https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers/posts/664832>
- [127] S. Diehl, P. Hartel, and P. Sestoft, “Abstract machines for programming language implementation,” *Future Generation Computer Systems*, vol. 16, no. 7, pp. 739–751, May 2000. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167739X99000886>

- [128] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, “Virtual machine showdown: Stack versus registers,” vol. 4, no. 4, pp. 2:1–2:36, 2008. [Online]. Available: <https://doi.org/10.1145/1328195.1328197>
- [129] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.
- [130] Labels as values (using the GNU compiler collection (GCC)). [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>
- [131] M. Jamieson. (2019) The Merlin abstract machine. [Online]. Available: <https://gitlab.com/mjamieson/merlin>
- [132] S. Pemberton and M. Daniels, *Pascal implementation*, combined ed ed. Chichester : E. Horwood ; New York : Halsted Press, 1982, type: Book; Book/Illustrated. [Online]. Available: <https://homepages.cwi.nl/~steven/pascal/book/pascalimplementation.html>
- [133] (2020) SB-projects - SB-assembler. [Online]. Available: <https://www.sbprojects.net/sbasml/>
- [134] S. L. P. Jones, “Implementing lazy functional languages on stock hardware: the spineless tagless G-machine,” vol. 2, no. 2, pp. 127–202, 1992, publisher: Cambridge University Press. [Online]. Available: <https://doi.org/10.1017/S0956796800000319>
- [135] R. Bornat, *Understanding and Writing Compilers*, 2020, ch. 13. [Online]. Available: <http://www.eis.mdx.ac.uk/staffpages/r.bornat/books/compiling.pdf>
- [136] (2020, May) Two-Minute fact sheet. [Online]. Available: https://www.eiffel.org/doc/eiffel/Two-Minute_fact_sheet
- [137] D. A. Terei and M. M. Chakravarty, “An LLVM backend for GHC,” *ACM SIGPLAN Notices*, vol. 45, no. 11, pp. 109–120, Sep. 2010. [Online]. Available: <https://doi.org/10.1145/2088456.1863538>
- [138] H. M. Gualandi and R. Ierusalimschy, “Pallene: a statically typed companion language for lua,” in *Proceedings of the XXII Brazilian Symposium on Programming Languages - SBLP '18*. ACM Press, pp. 19–26. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3264637.3264640>

- [139] 4.11. GHC Backends. [Online]. Available: https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/code-generators.html
- [140] R. Gibbs, M. Jamieson, and A. Tindale, “Anubis - A persistent Object-Oriented database programming language,” 1991, unpublished BSc. dissertation.
- [141] A. Dearle, G. N. C. Kirby, and R. Morrison, “Orthogonal persistence revisited,” in *Object Databases*, ser. Lecture Notes in Computer Science, M. C. Norrie and M. Grossniklaus, Eds. Springer, pp. 1–22.
- [142] M. P. Atkinson and R. Morrison, “Persistent first class procedures are enough,” in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, M. Joseph and R. Shyamasundar, Eds. Springer, 1984, pp. 223–240.
- [143] W. Koch and C. Oeters, “An Abstract ALGOL 68 Machine and its application in a machine independent compiler.” 01 1975, pp. 642–653.
- [144] Graphviz - Graph Visualization Software. [Online]. Available: <https://graphviz.org/>
- [145] R. Hunter, *The Essence of Compilers*. Hemel Hempstead, United Kingdom: Prentice Hall Europe, 1999, ch. 7, pp. 170–175.
- [146] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ch. 7, pp. 404–429.
- [147] (2020, Nov) 7. Simple statements — Python 3.9.0 documentation. [Online]. Available: https://docs.python.org/3/reference/simple_stmts.html#nonlocal
- [148] B. Stroustrup, *The C++ Programming Language, Third Edition*, 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ch. 10, pp. 230–231.
- [149] M. Lutz, *Programming Python: 4th edition*, ser. Nutshell handbook. O’Reilly Media, Incorporated, 2011. [Online]. Available: <https://www.oreilly.com/library/view/programming-python-4th/9781449398712/>
- [150] A. Davie and R. Morrison, *Recursive Descent Compiling*. Chichester, England: Ellis-Horwood, 1981, ch. 11, pp. 143–144.

- [151] R. Hyde, *Write Great Code, Volume 1, 2nd Edition: Understanding the Machine*. No Starch Press, 2020, ch. 11, pp. 317–320. [Online]. Available: <https://books.google.co.uk/books?id=z1v6DwAAQBAJ>
- [152] Labels as values (using the GNU compiler collection (GCC)). [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>
- [153] J. R. Levine, *Linkers and Loaders*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.
- [154] (2021) objcopy(1) - linux manual page. [Online]. Available: <https://man7.org/linux/man-pages/man1/objcopy.1.html>
- [155] J. Dongarra, M. A. Heroux, and P. Luszczek, “A new metric for ranking high-performance computing systems,” vol. 3, no. 1, pp. 30–35. [Online]. Available: <https://academic.oup.com/nsr/article-lookup/doi/10.1093/nsr/nwv084>
- [156] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, “The celerity open-source 511-core RISC-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips,” vol. 38, no. 2, pp. 30–41, 2018, conference Name: IEEE Micro.
- [157] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, “Overview of the SpiNNaker system architecture,” vol. 62, no. 12, pp. 2454–2467, 2013.
- [158] J. Seely, S. Erusalagandi, and J. Bethurem, “The MicroBlaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs,” p. 14, 2017.
- [159] C. Wolf. (2018, Nov.) PicoRV32: A Size-Optimized RISC-V CPU. Contribute to cliffordwolf/picorv32 development by creating an account on GitHub. [Online]. Available: <https://github.com/cliffordwolf/picorv32>
- [160] “AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-lite ACE and ACE-lite,” p. 306.

- [161] Timing analyzer clock analysis. [Online]. Available: <https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/timinganalyzer/clocking/tq-clock.html>
- [162] “Vivado design suite user guide: Design analysis and closure techniques (UG906),” p. 308.
- [163] (2019, Jan.) opensouce RISC-V implemented from scratch in one night!: darklife/darkriscv. [Online]. Available: <https://github.com/darklife/darkriscv>
- [164] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, 1st ed. Strawberry Canyon, 2017.
- [165] (2019) November 2019 — TOP500 Supercomputer Sites. [Online]. Available: <https://www.top500.org/lists/2019/11>
- [166] B. Catanzaro, *The SPARC Technical Papers*, ser. Sun technical reference library. Springer-Verlag, 1991. [Online]. Available: <https://books.google.co.uk/books?id=4dVQAAAAMAAJ>
- [167] Clang c language family frontend for LLVM. [Online]. Available: <https://clang.llvm.org/>
- [168] (2021) SDCC - small device c compiler. [Online]. Available: <http://sdcc.sourceforge.net/>
- [169] (2021) Open watcom. [Online]. Available: <http://openwatcom.org/>
- [170] Fibonacci sequence. [Online]. Available: https://rosettacode.org/wiki/Fibonacci_sequence#C
- [171] M. Jamieson. PhD source code repository. [Online]. Available: <https://gitlab.com/mjamieson/phd>
- [172] (2019) LINPACK_BENCH - The LINPACK Benchmark. [Online]. Available: https://people.math.sc.edu/Burkardt/c_src/linpack_bench/linpack_bench.c
- [173] W. Jackson and M. Dubaniowski, “Lu factorisation on xeon and xeon phi processors,” in *Parallel Computing: On the Road to Exascale*, ser. Advances in Parallel Computing, G. Joubert, H. Leather, M. Parsons, F. Peters, and M. Sawyer, Eds., vol. 27. IOS Press Ebooks, Mar. 2016, pp. 591 – 599.

- [174] J. Gilbreath, “A high-level language benchmark,” in *BYTE Magazine Volume 06 Number 09 - Artificial Intelligence*, pp. 180–198. [Online]. Available: <https://archive.org/details/byte-magazine-1981-09/page/n181/mode/2up>
- [175] The sieve of eratosthenes benchmarks. [Online]. Available: <https://www.keil.com/benchmarks/sieve.asp>
- [176] Why you should benchmark your embedded system | SEGGER blog. [Online]. Available: <https://testblog.segger.com/blog/2016/09/30/why-you-should-benchmark-your-embedded-system/>
- [177] Kaggle, “Data science bowl 2017,” <https://www.kaggle.com/c/data-science-bowl-2017>, 2017, accessed: 2018-07-25.
- [178] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ch. 9, pp. 517–518.
- [179] J. J. Gough and K. J. Gough, *Compiling for the .Net Common Language Runtime*. USA: Prentice Hall PTR, 2001, ch. 1.
- [180] Using the GNU compiler collection (GCC): Optimize options. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Optimize-Options.html#Optimize-Options>
- [181] ARCHER advanced MPI. [Online]. Available: http://www.archer.ac.uk/training/course-material/2016/09/160929_AdvMPI_EPCC/index.php
- [182] Ingenic semiconductor_m200 m150 JZ4780 JZ4775 JZ4760b. [Online]. Available: <http://www.ingenic.com.cn/en/?product/id/13.html>
- [183] (2013) NEON programmer’s guide. [Online]. Available: <https://developer.arm.com/documentation/den0018/latest>
- [184] (2020) Linker scripts - OSDev wiki. [Online]. Available: https://wiki.osdev.org/Linker_Scripts#KEEP
- [185] F. M. Castro, N. Guil, M. J. Marín-Jiménez, J. Pérez-Serrano, and M. Ujaldón, “Energy-based tuning of convolutional neural networks on multi-gpus,” *Concurrency and Computation: Practice and Experience*, p. e4786, 2018.

- [186] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, April 1965. [Online]. Available: <https://doi.org/10.1109/N-SSC.2006.4785860>
- [187] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” vol. 9, no. 5, pp. 256–268, 1974, conference Name: IEEE Journal of Solid-State Circuits. [Online]. Available: <https://doi.org/10.1109/JSSC.1974.1050511>
- [188] (2019) European Processor Initiative Accelerator. [Online]. Available: <https://www.european-processor-initiative.eu/accelerator/>
- [189] N. Brown and M. Jamieson. Vipera. [Online]. Available: <https://www.vipera.dev/>
- [190] “Eithne.” [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Eithne>
- [191] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, “Static type inference for ruby,” in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1859–1866. [Online]. Available: <https://doi.org/10.1145/1529282.1529700>
- [192] C. Reddy and U. Bondhugula, “Effective automatic computation placement and data allocation for parallelization of regular programs,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS ’14. ACM, pp. 13–22. [Online]. Available: <http://doi.acm.org/10.1145/2597652.2597673>
- [193] S. Kamil, D. Coetzee, and A. Fox, “Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization,” p. 7. [Online]. Available: <https://doi.org/10.25080/Majora-ebaa42b7-00f>
- [194] S. Sinha, M. Koedam, R. v. Wijk, A. Nelson, A. B. Nejad, M. Geilen, and K. Goossens, “Composable and predictable dynamic loading for time-critical partitioned systems,” in *2014 17th Euromicro Conference on Digital System Design*, pp. 285–292. [Online]. Available: <https://doi.org/10.1109/DSD.2014.40>
- [195] Y. Zhang, M. Yang, B. Zhou, Z. Yang, W. Zhang, and B. Zang, “Swift: A register-based JIT compiler for embedded JVMs,” in *Proceedings*

- of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. ACM, pp. 63–74. [Online]. Available: <http://doi.acm.org/10.1145/2151024.2151035>
- [196] S. R. Ladd, “Acovea: Analysis of compiler options via evolutionary algorithm,” *Describing the Evolutionary Algorithm*, 2007.
- [197] S. Zhong, Y. Shen, and F. Hao, “Tuning compiler optimization options via simulated annealing,” in *2009 Second International Conference on Future Information Technology and Management Engineering*, pp. 305–308.
- [198] Eval of julia code · the julia language. [Online]. Available: <https://docs.julialang.org/en/v1/devdocs/eval/>
- [199] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.
- [200] “MLIR-EmitC,” original-date: 2020-05-28T13:02:23Z. [Online]. Available: <https://github.com/impl130/mlir-emitc>
- [201] “The c++ for OpenCL 1.0 and 2021 programming language documentation,” p. 28. [Online]. Available: https://raw.githubusercontent.com/KhronosGroup/Khronosdotorg/main/api/opencl/assets/CXX_for_OpenCL.pdf



Fibonacci sequence code listings

```
1 ;-----
2 ;   fib.asm - Fibonacci test using basic stack-based opcodes
3 ;-----
4     .cr      merlin
5     .tf      fib.v12,bin
6     .dl 65566      ; VM version number
7     .dl END-6      ; Bytecode length
8
9     ALLOC 6*4      ; 8+16
10    LLI 50000000    ; Load integer literal onto stack
11    DUP            ; Duplicate TOS
12    JSR FIB        ; Jump to subroutine
13    LDI I          ; Load TOS with integer from memory @I
14    CALLX PRINT    ; Call #0 user defined PRINT function
15    HALT
16
17 FIB: STI COUNT      ; Store TOS integer value @COUNT
18     CALLX PRINT
19     SLI A, 0x01     ; Store integer literal in memory @A
20     SLI B, 0x01
```

```

21      SLI C, 0x01
22      SLI I, 0x00
23
24 LOOP:  LDI A          ; Load A onto stack
25          LDI B
26          ADD          ; Add TOS-1 and TOS, result on TOS
27          STI C        ; Store top of stack in memory @C
28          LDI B
29          STI A
30          LDI C
31          STI B
32          INCM I        ; Increment integer value in memory @I
33          LDI I
34          LDI COUNT
35          LT           ; Set FLAGS to true if TOS-1 < TOS
36          BEQ LOOP     ; Branch to LOOP if FLAGS is true
37          RTS          ; Return from subroutine
38
39 END .eq $    ; End marker for code size calc above
40      .sm RAM
41 A      .dl 0
42 B      .dl 0
43 C      .dl 0
44 I      .dl 0
45 COUNT  .dl 0
46 PRINT  .eq 0

```

Listing A.1: Stack-based Fibonacci Merlin assembly listing

```

1 ;-----
2 ;   fib_fast.asm - Fibonacci test using three address opcodes
3 ;-----
4      .lf      fib_fast.lst
5      .cr      merlin
6      .tf      fib_fast.v12,bin
7      .dl 65566      ; VM version number
8      .dl END-6      ; Bytecode length
9
10     ALLOC 5*4
11     LLI 50000000     ; Load integer literal onto stack
12     DUP             ; Duplicate TOS
13     JSR FIB         ; Jump to subroutine

```

```

14      LDI I          ; Load TOS with integer from memory @I
15      CALLX PRINT    ; Call #0 user defined PRINT function
16      HALT
17
18 FIB:   STI COUNT     ; Store TOS integer value in memory @COUNT
19      CALLX PRINT
20      SLI A, 0x01     ; Store integer literal in memory @A
21      SLI B, 0x01
22      SLI C, 0x01
23      SLI I, 0x00
24
25 LOOP:  ADDM C, A, B   ; Add values @A and @B, store in memory @C
26      MOV A, B        ; Copy integer in memory @B to memory @A
27      MOV B, C
28      INCM I          ; Increment integer value in memory @I
29      BLT I, COUNT, LOOP ; Branch to LOOP if value @I < @COUNT
30      RTS             ; Return from subroutine
31
32 END .eq $           ; End marker for code size calc above
33      .sm RAM
34 A      .dl 0
35 B      .dl 0
36 C      .dl 0
37 COUNT  .dl 0
38 I      .dl 0
39 PRINT  .eq 0

```

Listing A.2: Three-address Fibonacci Merlin assembly listing

```

1 ;-----
2 ;   fib_reg.asm - Fibonacci test using register opcodes
3 ;-----
4      .lf      fib_reg.lst
5      .cr      merlin
6      .tf      fib_reg.v12,bin
7      .dl 65566      ; VM version number
8      .dl END-6      ; Bytecode length
9
10     ALLOC 2*4
11     LLI 50000000    ; Load integer literal onto stack
12     DUP             ; Duplicate TOS
13     JSR FIB         ; Jump to subroutine

```

```

14         LDI I           ; Load TOS with integer from memory @I
15         CALLX PRINT     ; Call #0 user defined PRINT function
16         HALT
17
18 FIB:     STI COUNT       ; Store TOS integer value in memory @COUNT
19         CALLX PRINT
20         LLA 0x01         ; Load integer literal into accumulator
21         LLX 0x01
22         LLY 0x01
23         SLI I, 0x00      ; Store integer literal in memory @I
24
25 LOOP:    ADDXY           ; Add X & Y registers, store in accumulator
26         TYX             ; Transfer (copy) Y register to X register
27         TAY
28         INCM I          ; Increment integer value in memory @I
29         BLT I, COUNT, LOOP ; Branch to LOOP if value @I < @COUNT
30         RTS             ; Return from subroutine
31
32 END .eq $              ; End marker for code size calc above
33         .sm RAM
34 COUNT    .dl 0
35 I        .dl 0
36 PRINT    .eq 0

```

Listing A.3: Register-based Fibonacci Merlin assembly listing

```

1 #include <stdio.h>
2 #include <time.h>
3
4 float cpu_time ( void )
5 /*****
6  */
7 Purpose:
8     CPU_TIME returns the current reading on the CPU clock.
9
10 Discussion:
11     The CPU time measurements available through this routine are
12     often not very accurate. In some cases, the accuracy is no
13     better than a hundredth of a second.
14
15 Licensing:
16     This code is distributed under the GNU LGPL license.

```

```

17
18 Modified:
19     06 June 2005
20
21 Author:
22     John Burkardt
23
24 Parameters:
25     Output, float CPU_TIME, the current reading of the CPU clock,
26     in seconds.
27 */
28 {
29     float value;
30
31     value = ( float ) clock ( )
32             / ( float ) CLOCKS_PER_SEC;
33
34     return value;
35 }
36 /*****
37
38 void timestamp ( void )
39 /*****
40 /*
41 Purpose:
42     TIMESTAMP prints the current YMDHMS date as a time stamp.
43
44 Example:
45     31 May 2001 09:45:54 AM
46
47 Licensing:
48     This code is distributed under the GNU LGPL license.
49
50 Modified:
51     24 September 2003
52
53 Author:
54     John Burkardt
55
56 Parameters:
57     None
58 */

```

```

59 {
60 # define TIME_SIZE 40
61
62 static char time_buffer[TIME_SIZE];
63 const struct tm *tm;
64 size_t len;
65 time_t now;
66
67 now = time ( NULL );
68 tm = localtime ( &now );
69
70 len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p",
    tm );
71
72 printf ( "%s\n", time_buffer );
73
74 return;
75 # undef TIME_SIZE
76 }
77 /*****
78
79 int fib(float n){
80     int a=1, b=1, c=1, i=0;
81     while(i<n) {
82         c = a + b;
83         a = b;
84         b = c;
85         i++;
86     }
87     return i;
88 }
89
90 int main(void) {
91     float t1;
92     float t2;
93     float time;
94
95     timestamp ( );
96     t1 = cpu_time ( );
97     printf("%d\n", fib(50000000));
98     t2 = cpu_time ( );
99     time = t2 - t1;

```

```

100     printf("%fs\n", time);
101     timestamp ( );
102 }

```

Listing A.4: Fibonacci C listing

```

1  import time
2
3  def fib(n):
4      a=1
5      b=1
6      c=1
7      i=0
8      while i<n:
9          c = a + b
10         a = b
11         b = c
12         i+=1
13     return i
14
15 n = 50000000.0
16 times = 0.0
17
18 timestamp()
19 print(n, "\n")
20
21 t1 = cpu_time()
22 i=fib(n)
23 t2 = cpu_time()
24 time = t2 - t1
25
26 print(i, "\n")
27 print(time, "s\n")
28 timestamp ()

```

Listing A.5: Fibonacci ePython listing

```

1  native include(time)
2  #*****
3  # float cpu_time ( )
4  #*****
5  #

```

```

6 # Purpose:
7 #   CPU_TIME returns the current reading on the CPU clock.
8 #
9 # Discussion:
10 #   The CPU time measurements available through this routine are
11 #   often not very accurate. In some cases, the accuracy is no
12 #   better than a hundredth of a second.
13 #
14 # Licensing:
15 #   This code is distributed under the GNU LGPL license.
16 #
17 # Modified:
18 #   06 June 2005
19 #
20 # Author:
21 #   John Burkardt
22 #
23 # Parameters:
24 #   Output, float CPU_TIME, the current reading of the CPU clock,
25 #   in seconds.
26 def cpu_time():
27     return native(->float) clock( ) / native const(CLOCKS_PER_SEC)->
        float
28
29
30 *****
31 # void timestamp ( void )
32 *****
33 #
34 # Purpose:
35 #   TIMESTAMP prints the current YMDHMS date as a time stamp.
36 #
37 # Example:
38 #   31 May 2001 09:45:54 AM
39 #
40 # Licensing:
41 #   This code is distributed under the GNU LGPL license.
42 #
43 # Modified:
44 #   24 September 2003
45 #
46 # Author:

```

```

47 #     John Burkardt
48 #
49 #     Updated for ePython:
50 #     Maurice Jamieson
51 #
52 #     Parameters:
53 #     None
54 def timestamp():
55     # Turn off GCC compiler warnings as we're performing a number of
56     # implicit casts - store the previous diagnostic settings
57     native pragma 'GCC diagnostic push'
58     native pragma 'GCC diagnostic ignored "-Wformat"'
59     native pragma 'GCC diagnostic push'
60     native pragma 'GCC diagnostic ignored "-Wint-conversion"'
61     native pragma 'GCC diagnostic push'
62     native pragma 'GCC diagnostic ignored "-Wincompatible-pointer-
        types"'
63
64     TIME_SIZE = 40
65     time_buffer = native malloc(41)
66     now = native time ( native const(NULL) )
67     tm = native localtime ( &now )
68
69     len = native strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S
        %p", tm)
70
71     native printf ( "%s\n", time_buffer )
72
73     # Restore previous GCC diagnostic settings
74     native pragma 'GCC diagnostic pop'
75     native pragma 'GCC diagnostic pop'
76     native pragma 'GCC diagnostic pop'
77
78     return

```

Listing A.6: ePython time module listing

```

1 #include "fib_oly_extrns.h"
2
3 #define OLY_ENV_SIZE 16+3
4
5 void module(Memory *memory) {
6 #ifdef SYMBOL_TABLE

```

```

7 initialise_symbol_table(OLY_ENV_SIZE);
8 #endif
9 MKENV(memory, OLY_ENV_SIZE);
10 DECLL("cpu_time", ADDR(0), MKLAMBDA(oly_r$cpu_time$, 1));
11 DECLL("timestamp", ADDR(1), MKLAMBDA(oly_v$timestamp$, 6));
12 DECLL("fib", ADDR(2), MKLAMBDA(oly_i$fib$r, 6));
13 DECLR("n", ADDR(3), 50000000.000000);
14 DECLR("times", ADDR(4), 0.000000);
15 (MKBETA(DDL(ADDR(1))), APPLYV(DDL(ADDR(1))));
16 rtl_print(2, 7, LDR(ADDR(3)), 11, "\n");
17 DECLR("t1", ADDR(5), (MKBETA(DDL(ADDR(0))), APPLYR(DDL(ADDR(0)))));
18 DECLI("i", ADDR(6), (MKBETA(DDL(ADDR(2))), PARAMR(1, "n", LDR(ADDR(3))
    ), APPLYI(DDL(ADDR(2)))));
19 DECLR("t2", ADDR(7), (MKBETA(DDL(ADDR(0))), APPLYR(DDL(ADDR(0)))));
20 DECLR("time", ADDR(8), (LDR(ADDR(7)) - LDR(ADDR(5))));
21 rtl_print(2, 5, LDI(ADDR(6)), 11, "\n");
22 rtl_print(2, 7, LDR(ADDR(8)), 11, "s\n");
23 (MKBETA(DDL(ADDR(1))), APPLYV(DDL(ADDR(1))));
24 }

```

Listing A.7: Generated Olympus main file for ePython Fibonacci benchmark

```

1 #include "fib_oly_externs.h"
2
3 void oly_r$cpu_time$(Memory *memory, Object *self) {
4 RETR((EXPR(Real, clock(), /, Real, CLOCKS_PER_SEC)));
5 }
6
7 void oly_v$timestamp$(Memory *memory, Object *self) {
8 #pragma GCC diagnostic push
9 #pragma GCC diagnostic ignored "-Wformat"
10 #pragma GCC diagnostic push
11 #pragma GCC diagnostic ignored "-Wint-conversion"
12 #pragma GCC diagnostic push
13 #pragma GCC diagnostic ignored "-Wincompatible-pointer-types"
14 DECLI("TIME_SIZE", ADDR(1), 40);
15 DECLI("time_buffer", ADDR(2), malloc(41));
16 DECLI("now", ADDR(3), time(NULL));
17 DECLI("tm", ADDR(4), localtime((Pntr)(ADDR(3))));
18 DECLI("len", ADDR(5), strftime(LDI(ADDR(2)), LDI(ADDR(1)), "%d %B %Y
    %I:%M:%S %p", LDI(ADDR(4))));
19 printf("%s\n", LDI(ADDR(2)));
20 #pragma GCC diagnostic pop

```

```

21 #pragma GCC diagnostic pop
22 #pragma GCC diagnostic pop
23 RETV;
24 }
25
26 void oly_i$fib$r(Memory *memory, Object *self) {
27 DECLI("a", ADDR(2), 1);
28 DECLI("b", ADDR(3), 1);
29 DECLI("c", ADDR(4), 1);
30 DECLI("i", ADDR(5), 0);
31 while ((LDI(ADDR(5)) < LDR(ADDR(1)))) {
32 STI(ADDR(4), (LDI(ADDR(2)) + LDI(ADDR(3))));
33 STI(ADDR(2), LDI(ADDR(3)));
34 STI(ADDR(3), LDI(ADDR(4)));
35 STI(ADDR(5), (LDI(ADDR(5)) + 1));
36 };
37 RETI(LDI(ADDR(5)));
38 }

```

Listing A.8: Generated Olympus functions file for ePython Fibonacci benchmark

```

1 #ifndef FIB_OLY_EXTERNS_H
2 #define FIB_OLY_EXTERNS_H
3
4 /* System required and user requested includes in 'fib' */
5 #include "olympus.h"
6
7 #include "time.h"
8
9 /* User defined functions (including variants for polymorphism) in '
   fib' */
10 extern void oly_r$cpu_time$(Memory*, Object*);
11 extern void oly_v$timestamp$(Memory*, Object*);
12 extern void oly_i$fib$r(Memory*, Object*);
13
14 /* Native functions used by 'fib' */
15 extern Void rtl_print(Int, ...);
16
17 #endif /* FIB_OLY_EXTERNS_H */

```

Listing A.9: Generated Olympus external functions file for ePython Fibonacci benchmark

B

LINPACK benchmark code listings

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <string.h>
5 #include <time.h>
6 #include <unistd.h>
7
8 #include "eithne.h"
9 #include "kernels.h"
10
11 #define TARGET_ID 1
12
13 #define SLEEP_TIME 0000
14
15 int main ( );
16 double cpu_time ( );
17 float r8_abs ( float x );
18 float r8_epsilon ( );
19 float r8_max ( float x, float y );
20 float r8_random ( int izeed[4] );
```

```

21 float *r8mat_gen ( int lda, int n );
22 void timestamp ( );
23 void printarray(const char*, float*, int);
24
25 int info=1;
26 int job;
27
28 /*****
29 int main ( )
30 /*****
31 /*
32     Purpose:
33
34     MAIN is the main program for LINPACK_BENCH.
35
36     Discussion:
37
38     LINPACK_BENCH drives the float precision LINPACK benchmark
39     program.
40
41     Modified:
42
43     25 July 2008
44
45     Parameters:
46
47     N is the problem size.
48 */
49 {
50     float *a;
51     float a_max;
52     float *b;
53     float b_max;
54     float cray = 0.056;
55     float eps;
56     int i;
57     int *ipvt;
58     int j;
59     float ops;
60     float *resid;
61     float resid_max;
62     float residn;

```

```

62  float *rhs;
63  float t1;
64  float t2;
65  float time[6];
66  float total;
67  float *x;
68
69  EithneSharedMem eithne_buffer;
70  static EithneVariable variables[5];
71  eithne_buffer = EITHNE_ALLOC_MEM(sizeof(float)*N*LDA);
72
73  EITHNE_INIT_HOST(variables, HOST_ID, eithne_buffer +
    EITHNE_DATA_OFFSET, eithne_buffer);
74
75  /* This will load the code onto the cores */
76  EITHNE_INIT_CORES(1);
77  EITHNE_START_CORES(1);
78  usleep(SLEEP_TIME);
79
80  timestamp ( );
81  printf ( "\n" );
82  printf ( "LINPACK_BENCH\n" );
83  printf ( "  C version\n" );
84  printf ( "\n" );
85  printf ( "  The LINPACK benchmark.\n" );
86  printf ( "  Language: C\n" );
87  printf ( "  Datatype: Single precision real\n" );
88  printf ( "  Matrix order N                = %d\n", N );
89  printf ( "  Leading matrix dimension LDA = %d\n", LDA );
90
91  ops = (float) ( 2 * N * N * N ) / 3.0 + 2.0 * (float) ( N * N );
92  /*
93  Allocate space for arrays.
94  */
95  a = r8mat_gen ( LDA, N );
96  b = ( float * ) malloc ( N * sizeof ( float ) );
97  ipvt = ( int * ) malloc ( N * sizeof ( int ) );
98  resid = ( float * ) malloc ( N * sizeof ( float ) );
99  rhs = ( float * ) malloc ( N * sizeof ( float ) );
100 x = ( float * ) malloc ( N * sizeof ( float ) );
101
102  /*

```

```

103     We want to:
104
105     Define a, b, ipvt, job and result as Eithne variables in
        kernels.h
106
107     The following register the variables on the host side:
108     */
109     EITHNE_REGISTER_ARRAY(variables,A,EITHNE_FLOAT_ARRAY,a,N*LDA);
110     EITHNE_REGISTER_ARRAY(variables,B,EITHNE_FLOAT_ARRAY,b,N);
111     EITHNE_REGISTER_ARRAY(variables,IPVT,EITHNE_INTEGER_ARRAY,ipvt,N);
112     EITHNE_REGISTER_SCALAR(variables,JOB,EITHNE_INTEGER,job);
113     EITHNE_REGISTER_SCALAR(variables,RESULT,EITHNE_INTEGER,info);
114
115     a_max = 0.0;
116     for ( j = 0; j < N; j++ )
117     {
118         for ( i = 0; i < N; i++ )
119         {
120             a_max = r8_max ( a_max, a[i+j*LDA] );
121         }
122     }
123
124     for ( i = 0; i < N; i++ )
125     {
126         x[i] = 1.0;
127     }
128
129     for ( i = 0; i < N; i++ )
130     {
131         b[i] = 0.0;
132         for ( j = 0; j < N; j++ )
133         {
134             b[i] = b[i] + a[i+j*LDA] * x[j];
135         }
136     }
137
138     /* Kernel #1 - SGEFA */
139     /*
140         Variable types and sizes have been defined above -
141         (slices with offset and size can be sent /received by other
        functions)
142     */

```

```

143
144 EITHNE_SEND(variables, TARGET_ID, A);
145
146 t1 = cpu_time ( );
147
148 /* The kernel listener will execute the requested function */
149 EITHNE_EXECUTE(TARGET_ID, SGEFA);
150
151 t2 = cpu_time ( );
152
153 /* Output variables from SGEFA */
154 EITHNE_RECV(variables, TARGET_ID, A);
155 EITHNE_RECV(variables, TARGET_ID, IPVT);
156 EITHNE_RECV(variables, TARGET_ID, RESULT);
157
158 if ( info != 0 )
159 {
160     printf ( "\n" );
161     printf ( "LINPACK_BENCH - Fatal error!\n" );
162     printf ( "  The matrix A is apparently singular.\n" );
163     printf ( "  Abnormal end of execution.\n" );
164     return 1;
165 }
166
167 time[0] = t2 - t1;
168
169 job = 0;
170
171 /* Kernel #2 - SGESL */
172 EITHNE_SEND(variables, TARGET_ID, A);
173 EITHNE_SEND(variables, TARGET_ID, B);
174 EITHNE_SEND(variables, TARGET_ID, IPVT);
175 EITHNE_SEND(variables, TARGET_ID, JOB);
176
177 t1 = cpu_time ( );
178
179 /* The kernel listener will execute the requested function */
180 EITHNE_EXECUTE(TARGET_ID, SGESL);
181
182 t2 = cpu_time ( );
183
184 /* Output variables from SGESL */

```

```
185 EITHNE_RECV(variables, TARGET_ID, B);
186
187 time[1] = t2 - t1;
188
189 total = time[0] + time[1];
190
191 free ( a );
192 /*
193  Compute a residual to verify results.
194 */
195 a = r8mat_gen ( LDA, N );
196
197 for ( i = 0; i < N; i++ )
198 {
199     x[i] = 1.0;
200 }
201
202 for ( i = 0; i < N; i++ )
203 {
204     rhs[i] = 0.0;
205     for ( j = 0; j < N; j++ )
206     {
207         rhs[i] = rhs[i] + a[i+j*LDA] * x[j];
208     }
209 }
210
211 for ( i = 0; i < N; i++ )
212 {
213     resid[i] = -rhs[i];
214     for ( j = 0; j < N; j++ )
215     {
216         resid[i] = resid[i] + a[i+j*LDA] * b[j];
217     }
218 }
219
220 resid_max = 0.0;
221 for ( i = 0; i < N; i++ )
222 {
223     resid_max = r8_max ( resid_max, r8_abs ( resid[i] ) );
224 }
225
226 b_max = 0.0;
```

```

227  for ( i = 0; i < N; i++ )
228  {
229      b_max = r8_max ( b_max, r8_abs ( b[i] ) );
230  }
231
232  eps = r8_epsilon ( );
233
234  residn = resid_max / ( float ) N / a_max / b_max / eps;
235
236  time[2] = total;
237  if ( 0.0 < total )
238  {
239      time[3] = ops / ( 1.0E+06 * total );
240  }
241  else
242  {
243      time[3] = -1.0;
244  }
245  time[4] = 2.0 / time[3];
246  time[5] = total / cray;
247
248  printf ( "\n" );
249  printf ( "      Norm. Resid      Resid      MACHEP      X
250  [1]      X[N]\n" );
251  printf ( "\n" );
252  printf ( " %14f %14f %14e %14f %14f\n", residn, resid_max,
253  eps, b[0], b[N-1] );
254  printf ( "\n" );
255  printf ( "      Factor      Solve      Total      MFLOPS      Unit
256  Cray-Ratio\n" );
257  printf ( "\n" );
258  printf ( " %9f %9f %9f %9f %9f %9f\n",
259  time[0], time[1], time[2], time[3], time[4], time[5] );
260
261  /*
262  Terminate.
263  */
264  EITHNE_FINALISE_CORES;
265
266  free ( a );
267  free ( b );
268  free ( ipvt );

```

```

266   free ( resid );
267   free ( rhs );
268   free ( x );
269
270   printf ( "\n" );
271   printf ( "LINPACK_BENCH\n" );
272   printf ( "   Normal end of execution.\n" );
273
274   printf ( "\n" );
275   timestamp ( );
276
277   return 0;
278 # undef LDA
279 # undef N
280 }
281
282 /*****
283 double cpu_time ( void )
284 /*****
285 /*
286 Purpose:
287
288     CPU_TIME returns the current reading on the CPU clock.
289
290 Discussion:
291
292     The CPU time measurements available through this routine are
293     often
294     not very accurate.  In some cases, the accuracy is no better
295     than
296     a hundredth of a second.
297
298 Licensing:
299
300     This code is distributed under the GNU LGPL license.
301
302 Modified:
303
304     06 June 2005
305
306 Author:

```

```

306     John Burkardt
307
308     Parameters:
309
310         Output, float CPU_TIME, the current reading of the CPU clock, in
           seconds.
311 */
312 {
313     float value;
314
315     value = ( float ) clock ( )
316             / ( float ) CLOCKS_PER_SEC;
317
318     return value;
319 }
320
321 /*****
322 void timestamp ( void )
323 *****/
324 /*
325 Purpose:
326
327     TIMESTAMP prints the current YMDHMS date as a time stamp.
328
329 Example:
330
331     31 May 2001 09:45:54 AM
332
333 Licensing:
334
335     This code is distributed under the GNU LGPL license.
336
337 Modified:
338
339     24 September 2003
340
341 Author:
342
343     John Burkardt
344
345 Parameters:
346

```

```

347     None
348 */
349 {
350 # define TIME_SIZE 40
351
352 static char time_buffer[TIME_SIZE];
353 const struct tm *tm;
354 size_t len;
355 time_t now;
356
357 now = time ( NULL );
358 tm = localtime ( &now );
359
360 len = strftime(time_buffer,TIME_SIZE,"%d %B %Y %I:%M:%S %p",tm );
361
362 printf ( "%s\n", time_buffer );
363
364 return;
365 # undef TIME_SIZE
366 }
367
368 /*****
369 float r8_abs ( float x )
370 /*****
371 /*
372 Purpose:
373
374     R8_ABS returns the absolute value of a R8.
375
376 Modified:
377
378     02 April 2005
379
380 Author:
381
382     John Burkardt
383
384 Parameters:
385
386     Input, float X, the quantity whose absolute value is desired.
387
388     Output, float R8_ABS, the absolute value of X.

```

```

389 */
390 {
391     float value;
392
393     if ( 0.0 <= x )
394     {
395         value = x;
396     }
397     else
398     {
399         value = -x;
400     }
401     return value;
402 }
403
404 /*****
405 float r8_epsilon ( )
406 /*****/
407 /*
408 Purpose:
409
410     R8_EPSILON returns the R8 round off unit.
411
412 Discussion:
413
414     R8_EPSILON is a number R which is a power of 2 with the property
415     that,
416     to the precision of the computer's arithmetic,
417         1 < 1 + R
418     but
419         1 = ( 1 + R / 2 )
420
421 Licensing:
422
423     This code is distributed under the GNU LGPL license.
424
425 Modified:
426
427     01 September 2012
428
429 Author:

```

```

430     John Burkardt
431
432     Parameters:
433
434         Output, float R8_EPSILON, the R8 round-off unit.
435 */
436 {
437     const float value = 2.220446049250313E-016;
438
439     return value;
440 }
441
442 /*****
443 float r8_max ( float x, float y )
444 /*****
445 /*
446     Purpose:
447
448         R8_MAX returns the maximum of two R8's.
449
450     Modified:
451
452         18 August 2004
453
454     Author:
455
456         John Burkardt
457
458     Parameters:
459
460         Input, float X, Y, the quantities to compare.
461
462         Output, float R8_MAX, the maximum of X and Y.
463 */
464 {
465     float value;
466
467     if ( y < x )
468     {
469         value = x;
470     }
471     else

```

```

472 {
473     value = y;
474 }
475 return value;
476 }
477
478 /*****
479 float r8_random ( int izeed[4] )
480 /*****
481 /*
482 Purpose:
483
484     R8_RANDOM returns a uniformly distributed random number between
485     0 and 1.
486
487 Discussion:
488
489     This routine uses a multiplicative congruential method with
490     modulus
491     2**48 and multiplier 33952834046453 (see G.S.Fishman,
492     'Multiplicative congruential random number generators with
493     modulus
494     2**b: an exhaustive analysis for b = 32 and a partial analysis
495     for
496     b = 48', Math. Comp. 189, pp 331-344, 1990).
497
498     48-bit integers are stored in 4 integer array elements with 12
499     bits
500     per element. Hence the routine is portable across machines with
501     integers of 32 bits or more.
502
503 Parameters:
504
505     Input/output, integer ISEED(4).
506     On entry, the seed of the random number generator; the array
507     elements must be between 0 and 4095, and ISEED(4) must be odd.
508     On exit, the seed is updated.
509
510     Output, float R8_RANDOM, the next pseudorandom number.
511 */
512 {
513     int ipw2 = 4096;

```

```

509  int it1;
510  int it2;
511  int it3;
512  int it4;
513  int m1 = 494;
514  int m2 = 322;
515  int m3 = 2508;
516  int m4 = 2549;
517  float one = 1.0;
518  float r = 1.0 / 4096.0;
519  float value;
520  /*
521  Multiply the seed by the multiplier modulo 2**48.
522  */
523  it4 = iseed[3] * m4;
524  it3 = it4 / ipw2;
525  it4 = it4 - ipw2 * it3;
526  it3 = it3 + iseed[2] * m4 + iseed[3] * m3;
527  it2 = it3 / ipw2;
528  it3 = it3 - ipw2 * it2;
529  it2 = it2 + iseed[1] * m4 + iseed[2] * m3 + iseed[3] * m2;
530  it1 = it2 / ipw2;
531  it2 = it2 - ipw2 * it1;
532  it1 = it1 + iseed[0] * m4 + iseed[1] * m3 + iseed[2] * m2 + iseed
    [3] * m1;
533  it1 = ( it1 % ipw2 );
534  /*
535  Return updated seed
536  */
537  iseed[0] = it1;
538  iseed[1] = it2;
539  iseed[2] = it3;
540  iseed[3] = it4;
541  /*
542  Convert 48-bit integer to a real number in the interval (0,1)
543  */
544  value =
545      r * ( ( float ) ( it1 )
546      + r * ( ( float ) ( it2 )
547      + r * ( ( float ) ( it3 )
548      + r * ( ( float ) ( it4 ) ) ) ) );
549

```

```

550     return value;
551 }
552
553 /*****
554 float *r8mat_gen ( int lda, int n )
555 /*****
556 /*
557     Purpose:
558
559     R8MAT_GEN generates a random R8MAT.
560
561     Modified:
562
563     06 June 2005
564
565     Parameters:
566
567     Input, integer LDA, the leading dimension of the matrix.
568
569     Input, integer N, the order of the matrix.
570
571     Output, float R8MAT_GEN[LDA*N], the N by N matrix.
572 */
573 {
574     float *a;
575     int i;
576     int init[4] = { 1, 2, 3, 1325 };
577     int j;
578
579     a = ( float * ) malloc ( lda * n * sizeof ( float ) );
580
581     for ( j = 1; j <= n; j++ )
582     {
583         for ( i = 1; i <= n; i++ )
584         {
585             a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5;
586         }
587     }
588
589     return a;
590 }

```

Listing B.1: LINPACK C benchmark with Eithne API calls

```

1 #include "eithne.h"
2 #include "kernels.h"
3
4 static EithneVariable variables[5];
5
6 static float a[N*LDA];
7 static float b[N];
8 static int ipvt[N];
9 static int job;
10 static int info=6;
11
12 static void saxpy ( int n, float da, float dx[], int incx, float dy
    [], int incy );
13 static float ddot ( int n, float dx[], int incx, float dy[], int
    incy );
14 static void sgefa(void);
15 static void sgesl(void);
16 static void sscal ( int n, float sa, float x[], int incx );
17 static int isamax ( int n, float dx[], int incx );
18 static float r8_abs ( float x );
19
20 void kernel_init(EithneTargetId id, EithneSharedMem buffer) {
21     EithneKernel kernels[] = { sgefa, sgesl };
22
23     EITHNE_INIT_DEVICE(variables, id, buffer + EITHNE_DATA_OFFSET,
        buffer, kernels);
24
25     EITHNE_REGISTER_ARRAY(variables, A, EITHNE_FLOAT_ARRAY, a, N*LDA);
26     EITHNE_REGISTER_ARRAY(variables, B, EITHNE_FLOAT_ARRAY, b, N);
27     EITHNE_REGISTER_ARRAY(variables, IPVT, EITHNE_INTEGER_ARRAY, ipvt,
        N);
28     EITHNE_REGISTER_SCALAR(variables, JOB, EITHNE_INTEGER, job);
29     EITHNE_REGISTER_SCALAR(variables, RESULT, EITHNE_INTEGER, info);
30
31     EITHNE_START_LISTENER;
32 }
33
34 /*****/

```

```
35 static void saxpy ( int n, float da, float dx[], int incx, float dy
    [], int incy )
36 /*****
37 /*
38 Purpose:
39
40     SAXPY computes constant times a vector plus a vector.
41
42 Discussion:
43
44     This routine uses unrolled loops for increments equal to one.
45
46 Modified:
47
48     30 March 2007
49
50 Author:
51
52     FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
    Pete Stewart.
53     C version by John Burkardt
54
55 Reference:
56
57     Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
58     LINPACK User's Guide,
59     SIAM, 1979.
60
61     Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
62     Basic Linear Algebra Subprograms for Fortran Usage,
63     Algorithm 539,
64     ACM Transactions on Mathematical Software,
65     Volume 5, Number 3, September 1979, pages 308-323.
66
67 Parameters:
68
69     Input, int N, the number of elements in DX and DY.
70
71     Input, float DA, the multiplier of DX.
72
73     Input, float DX[*], the first vector.
74
```

```
75     Input, int INCX, the increment between successive entries of DX
76
77     Input/output, float DY[*], the second vector.
78     On output, DY[*] has been replaced by  $DY[*] + DA * DX[*]$ .
79
80     Input, int INCY, the increment between successive entries of DY
81 */
82 {
83     int i;
84     int ix;
85     int iy;
86     int m;
87
88     if ( n <= 0 )
89     {
90         return;
91     }
92
93     if ( da == 0.0 )
94     {
95         return;
96     }
97 /*
98     Code for unequal increments or equal increments
99     not equal to 1.
100 */
101     if ( incx != 1 || incy != 1 )
102     {
103         if ( 0 <= incx )
104         {
105             ix = 0;
106         }
107         else
108         {
109             ix = ( - n + 1 ) * incx;
110         }
111
112         if ( 0 <= incy )
113         {
114             iy = 0;
115         }
116         else
```

```

117     {
118         iy = ( - n + 1 ) * incy;
119     }
120
121     for ( i = 0; i < n; i++ )
122     {
123         dy[iy] = dy[iy] + da * dx[ix];
124         ix = ix + incx;
125         iy = iy + incy;
126     }
127 }
128 /*
129  Code for both increments equal to 1.
130 */
131 else
132 {
133     m = n % 4;
134
135     for ( i = 0; i < m; i++ )
136     {
137         dy[i] = dy[i] + da * dx[i];
138     }
139
140     for ( i = m; i < n; i = i + 4 )
141     {
142         dy[i ] = dy[i ] + da * dx[i ];
143         dy[i+1] = dy[i+1] + da * dx[i+1];
144         dy[i+2] = dy[i+2] + da * dx[i+2];
145         dy[i+3] = dy[i+3] + da * dx[i+3];
146     }
147 }
148 return;
149 }
150 /*****
151 static float ddot ( int n, float dx[], int incx, float dy[], int
    incy )
152 /*****
153 /*
154 Purpose:
155
156     DDOT forms the dot product of two vectors.
157

```

```
158 Discussion:
159
160     This routine uses unrolled loops for increments equal to one.
161
162 Modified:
163
164     30 March 2007
165
166 Author:
167
168     FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
169     Pete Stewart.
170
171     C version by John Burkardt
172
173 Reference:
174
175     Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
176     LINPACK User's Guide,
177     SIAM, 1979.
178
179     Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
180     Basic Linear Algebra Subprograms for Fortran Usage,
181     Algorithm 539,
182     ACM Transactions on Mathematical Software,
183     Volume 5, Number 3, September 1979, pages 308-323.
184
185 Parameters:
186
187     Input, int N, the number of entries in the vectors.
188
189     Input, float DX[*], the first vector.
190
191     Input, int INCX, the increment between successive entries in DX
192
193     Input, float DY[*], the second vector.
194
195     Input, int INCY, the increment between successive entries in DY
196
197     Output, float DDOT, the sum of the product of the corresponding
198     entries of DX and DY.
199 */
200 {
```

```
199  float dtemp;
200  int i;
201  int ix;
202  int iy;
203  int m;
204
205  dtemp = 0.0;
206
207  if ( n <= 0 )
208  {
209      return dtemp;
210  }
211  /*
212   Code for unequal increments or equal increments
213   not equal to 1.
214  */
215  if ( incx != 1 || incy != 1 )
216  {
217      if ( 0 <= incx )
218      {
219          ix = 0;
220      }
221      else
222      {
223          ix = ( - n + 1 ) * incx;
224      }
225
226      if ( 0 <= incy )
227      {
228          iy = 0;
229      }
230      else
231      {
232          iy = ( - n + 1 ) * incy;
233      }
234
235      for ( i = 0; i < n; i++ )
236      {
237          dtemp = dtemp + dx[ix] * dy[iy];
238          ix = ix + incx;
239          iy = iy + incy;
240      }
```

```

241 }
242 /*
243 Code for both increments equal to 1.
244 */
245 else
246 {
247     m = n % 5;
248
249     for ( i = 0; i < m; i++ )
250     {
251         dtemp = dtemp + dx[i] * dy[i];
252     }
253
254     for ( i = m; i < n; i = i + 5 )
255     {
256         dtemp = dtemp + dx[i ] * dy[i ]
257                     + dx[i+1] * dy[i+1]
258                     + dx[i+2] * dy[i+2]
259                     + dx[i+3] * dy[i+3]
260                     + dx[i+4] * dy[i+4];
261     }
262 }
263 return dtemp;
264 }
265
266 /*****
267 static void sgefa (void)
268 /*****
269 /*
270 Purpose:
271
272     SGEFA factors a real general matrix.
273
274 Modified:
275
276     16 May 2005
277
278 Author:
279
280     C version by John Burkardt.
281
282 Reference:

```

```
283
284     Jack Dongarra, Cleve Moler, Jim Bunch and Pete Stewart,
285     LINPACK User's Guide,
286     SIAM, (Society for Industrial and Applied Mathematics),
287     3600 University City Science Center,
288     Philadelphia, PA, 19104-2688.
289     ISBN 0-89871-172-X
290
291     Parameters:
292
293     Input/output, float A[LDA*N].
294     On input, the matrix to be factored.
295     On output, an upper triangular matrix and the multipliers used
296     to obtain
297     it. The factorization can be written  $A=L*U$ , where L is a
298     product of
299     permutation and unit lower triangular matrices, and U is upper
300     triangular.
301
302     Input, int LDA, the leading dimension of A.
303
304     Input, int N, the order of the matrix A.
305
306     Output, int IPVT[N], the pivot indices.
307
308     Output, int SGEFA, singularity indicator.
309     0, normal value.
310     K, if  $U(K,K) == 0$ . This is not an error condition for this
311     subroutine,
312     but it does indicate that SGESL or DGEDI will divide by zero if
313     called.
314     Use RCOND in SGECO for a reliable indication of singularity.
315 */
316 {
317     int j;
318     int k;
319     int l;
320     float t;
321     /* Originally function parameters */
322     int lda=LDA;
323     int n=N;
```

```

320  /*
321  Gaussian elimination with partial pivoting.
322  */
323  info = 0;
324
325  for ( k = 1; k <= n-1; k++ )
326  {
327  /*
328  Find L = pivot index.
329  */
330      l = isamax ( n-k+1, a+(k-1)+(k-1)*lda, 1 ) + k - 1;
331      ipvt[k-1] = l;
332  /*
333  Zero pivot implies this column already triangularized.
334  */
335      if ( a[l-1+(k-1)*lda] == 0.0 )
336      {
337          info = k;
338          continue;
339      }
340  /*
341  Interchange if necessary.
342  */
343      if ( l != k )
344      {
345          t = a[l-1+(k-1)*lda];
346          a[l-1+(k-1)*lda] = a[k-1+(k-1)*lda];
347          a[k-1+(k-1)*lda] = t;
348      }
349  /*
350  Compute multipliers.
351  */
352      t = -1.0 / a[k-1+(k-1)*lda];
353
354      sscal ( n-k, t, a+k+(k-1)*lda, 1 );
355  /*
356  Row elimination with column indexing.
357  */
358      for ( j = k+1; j <= n; j++ )
359      {
360          t = a[l-1+(j-1)*lda];
361          if ( l != k )

```

```

362     {
363         a[l-1+(j-1)*lda] = a[k-1+(j-1)*lda];
364         a[k-1+(j-1)*lda] = t;
365     }
366     saxpy ( n-k, t, a+k+(k-1)*lda, 1, a+k+(j-1)*lda, 1 );
367 }
368
369 }
370
371 ipvt[n-1] = n;
372
373 if ( a[n-1+(n-1)*lda] == 0.0 )
374 {
375     info = n;
376 }
377
378 return;
379 }
380
381 /*****
382 static void sgesl (void)
383 *****/
384 /*
385 Purpose:
386
387     SGESL solves a real general linear system  $A * X = B$ .
388
389 Discussion:
390
391     SGESL can solve either of the systems  $A * X = B$  or  $A' * X = B$ .
392
393     The system matrix must have been factored by SGECO or SGEFA.
394
395     A division by zero will occur if the input factor contains a
396     zero on the diagonal. Technically this indicates singularity
397     but it is often caused by improper arguments or improper
398     setting of LDA. It will not occur if the subroutines are
399     called correctly and if SGECO has set  $0.0 < RCOND$ 
400     or SGEFA has set  $INFO == 0$ .
401
402 Modified:
403

```

```

404     16 May 2005
405
406     Author:
407
408     C version by John Burkardt.
409
410     Reference:
411
412     Jack Dongarra, Cleve Moler, Jim Bunch and Pete Stewart,
413     LINPACK User's Guide,
414     SIAM, (Society for Industrial and Applied Mathematics),
415     3600 University City Science Center,
416     Philadelphia, PA, 19104-2688.
417     ISBN 0-89871-172-X
418
419     Parameters:
420
421     Input, float A[LDA*N], the output from SGECO or SGEFA.
422
423     Input, int LDA, the leading dimension of A.
424
425     Input, int N, the order of the matrix A.
426
427     Input, int IPVT[N], the pivot vector from SGECO or SGEFA.
428
429     Input/output, float B[N].
430     On input, the right hand side vector.
431     On output, the solution vector.
432
433     Input, int JOB.
434     0, solve  $A * X = B$ ;
435     nonzero, solve  $A' * X = B$ .
436 */
437 {
438     int k;
439     int l;
440     float t;
441     /* Originally function parameters */
442     int lda=LDA;
443     int n=N;
444
445     /*

```

```

446  Solve A * X = B.
447  */
448  if ( job == 0 )
449  {
450      for ( k = 1; k <= n-1; k++ )
451      {
452          l = ipvt[k-1];
453          t = b[l-1];
454
455          if ( l != k )
456          {
457              b[l-1] = b[k-1];
458              b[k-1] = t;
459          }
460
461          saxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 );
462
463      }
464
465      for ( k = n; 1 <= k; k-- )
466      {
467          b[k-1] = b[k-1] / a[k-1+(k-1)*lda];
468          t = -b[k-1];
469          saxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 );
470      }
471  }
472  /*
473  Solve A' * X = B.
474  */
475  else
476  {
477      for ( k = 1; k <= n; k++ )
478      {
479          t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 );
480          b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
481      }
482
483      for ( k = n-1; 1 <= k; k-- )
484      {
485          b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 );
486          l = ipvt[k-1];
487

```

```
488     if ( 1 != k )
489     {
490         t = b[l-1];
491         b[l-1] = b[k-1];
492         b[k-1] = t;
493     }
494 }
495 }
496
497 return;
498 }
499
500 /*****
501 static void sscal ( int n, float sa, float x[], int incx )
502 /*****
503 /*
504 Purpose:
505
506     SSCAL scales a vector by a constant.
507
508 Modified:
509
510     30 March 2007
511
512 Author:
513
514     FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
515     Pete Stewart.
516     C version by John Burkardt
517
518 Reference:
519
520     Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
521     LINPACK User's Guide,
522     SIAM, 1979.
523
524     Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
525     Basic Linear Algebra Subprograms for Fortran Usage,
526     Algorithm 539,
527     ACM Transactions on Mathematical Software,
528     Volume 5, Number 3, September 1979, pages 308-323.
```

```
529 Parameters:
530
531     Input, int N, the number of entries in the vector.
532
533     Input, float SA, the multiplier.
534
535     Input/output, float X[*], the vector to be scaled.
536
537     Input, int INCX, the increment between successive entries of X.
538 */
539 {
540     int i;
541     int ix;
542     int m;
543
544     if ( n <= 0 )
545     {
546     }
547     else if ( incx == 1 )
548     {
549         m = n % 5;
550
551         for ( i = 0; i < m; i++ )
552         {
553             x[i] = sa * x[i];
554         }
555
556         for ( i = m; i < n; i = i + 5 )
557         {
558             x[i]    = sa * x[i];
559             x[i+1] = sa * x[i+1];
560             x[i+2] = sa * x[i+2];
561             x[i+3] = sa * x[i+3];
562             x[i+4] = sa * x[i+4];
563         }
564     }
565     else
566     {
567         if ( 0 <= incx )
568         {
569             ix = 0;
570         }
```

```

571     else
572     {
573         ix = ( - n + 1 ) * incx;
574     }
575
576     for ( i = 0; i < n; i++ )
577     {
578         x[ix] = sa * x[ix];
579         ix = ix + incx;
580     }
581 }
582 return;
583 }
584
585 /*****
586 static int isamax ( int n, float dx[], int incx )
587 /*****
588 /*
589 Purpose:
590
591     ISAMAX finds the index of the vector element of maximum absolute
592     value.
593
594 Discussion:
595
596     WARNING: This index is a 1-based index, not a 0-based index!
597
598 Modified:
599
600     30 March 2007
601
602 Author:
603
604     FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
605     Pete Stewart.
606     C version by John Burkardt
607
608 Reference:
609
610     Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
611     LINPACK User's Guide,
612     SIAM, 1979.

```

```
611
612     Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
613     Basic Linear Algebra Subprograms for Fortran Usage,
614     Algorithm 539,
615     ACM Transactions on Mathematical Software,
616     Volume 5, Number 3, September 1979, pages 308-323.
617
618     Parameters:
619
620     Input, int N, the number of entries in the vector.
621
622     Input, float X[*], the vector to be examined.
623
624     Input, int INCX, the increment between successive entries of SX
625
626     Output, int ISAMAX, the index of the element of maximum
627     absolute value.
628 */
629 {
630     float smax;
631     int i;
632     int ix;
633     int value;
634
635     value = 0;
636
637     if ( n < 1 || incx <= 0 )
638     {
639         return value;
640     }
641
642     value = 1;
643
644     if ( n == 1 )
645     {
646         return value;
647     }
648
649     if ( incx == 1 )
650     {
651         smax = r8_abs ( dx[0] );
652
```

```

653     for ( i = 1; i < n; i++ )
654     {
655         if ( smax < r8_abs ( dx[i] ) )
656         {
657             value = i + 1;
658             smax = r8_abs ( dx[i] );
659         }
660     }
661 }
662 else
663 {
664     ix = 0;
665     smax = r8_abs ( dx[0] );
666     ix = ix + incx;
667
668     for ( i = 1; i < n; i++ )
669     {
670         if ( smax < r8_abs ( dx[ix] ) )
671         {
672             value = i + 1;
673             smax = r8_abs ( dx[ix] );
674         }
675         ix = ix + incx;
676     }
677 }
678
679 return value;
680 }
681
682 /*****/
683 static float r8_abs ( float x )
684 /*****/
685 /*
686 Purpose:
687
688     R8_ABS returns the absolute value of a R8.
689
690 Modified:
691
692     02 April 2005
693
694 Author:

```

```

695
696     John Burkardt
697
698     Parameters:
699
700         Input, float X, the quantity whose absolute value is desired.
701
702         Output, float R8_ABS, the absolute value of X.
703 */
704 {
705     float value;
706
707     if ( 0.0 <= x )
708     {
709         value = x;
710     }
711     else
712     {
713         value = -x;
714     }
715     return value;
716 }

```

Listing B.2: LINPACK C benchmark kernel file

```

1 import linpack_bench_externs
2 import time
3
4 #*****
5 def main():
6 #*****
7 #
8 #   Purpose:
9 #
10 #       MAIN is the main program for LINPACK_BENCH.
11 #
12 #   Discussion:
13 #
14 #       LINPACK_BENCH drives the double precision LINPACK benchmark
15 #       program.
16 #
17 #   Modified:

```

```

17 #
18 #     25 July 2008
19 #
20 # Parameters:
21 #
22 #     N is the problem size.
23 #
24 N = 1000
25 LDA = N + 1
26 cray = 0.056
27 times = [0.0]*6
28
29 realsize = native sizeof(cray)
30 if realsize == 4:
31     precision = "Single"
32 else:
33     precision = "Double"
34
35 timestamp ( )
36 print ( "\nLINPACK_BENCH\n" )
37 print ( "  vPython (Olympus) version\n" )
38 print ( "\n  The LINPACK benchmark.\n" )
39 print ( "  Language: Python\n" )
40 print ( "  Datatype: ", precision, " precision real\n" )
41 print ( "  Matrix order N              = ", N, "\n" )
42 print ( "  Leading matrix dimension LDA = ", LDA, "\n" )
43
44 ops = ( 2 * N * N * N ) / 3.0 + 2.0 * ( N * N )
45
46 #
47 # Allocate space for arrays.
48 #
49 a = [0.0] * LDA * N
50 r8mat_gen ( a, LDA, N )
51
52 b = [0.0] * N
53 ipvt = [0] * N
54 resid = [0.0] * N
55 rhs = [0.0] * N
56 x = [0.0] * N
57
58 a_max = 0.0

```

```

59  j = 0
60  while j < N:
61      i = 0
62      while i < N:
63          a_max = r8_max ( a_max, a[i+j*LDA] )
64          i += 1
65      j += 1
66
67  i = 0
68  while i < N:
69      x[i] = 1.0
70      i += 1
71
72  i = 0
73  while i < N:
74      b[i] = 0.0
75      j = 0
76      while j < N:
77          b[i] = b[i] + a[i+j*LDA] * x[j]
78          j += 1
79      i += 1
80
81  #t1 = [0.0]
82  t1 = cpu_time ( )
83
84  info = dgefa ( a, LDA, N, ipvt )
85
86  if ( info != 0 ):
87      print ( "\nLINPACK_BENCH - Fatal error!\n" )
88      print ( "  The matrix A is apparently singular.\n" )
89      print ( "  Abnormal end of execution.\n" )
90      return 1
91
92  #t2 = [0.0]
93  t2 = cpu_time ( )
94  times[0] = t2 - t1
95
96  t1 = cpu_time ( )
97
98  job = 0
99
100 dgesl ( a, LDA, N, ipvt, b, job )

```

```
101
102     t2 = cpu_time ( )
103
104     times[1] = t2 - t1
105
106     total = times[0] + times[1]
107     #
108     #   Compute a residual to verify results.
109     #
110     r8mat_gen ( a, LDA, N )
111
112     i = 0
113     while i < N:
114
115         x[i] = 1.0
116         i += 1
117
118     i = 0
119     while i < N:
120         rhs[i] = 0.0
121         j = 0
122         while j < N:
123             rhs[i] = rhs[i] + a[i+j*LDA] * x[j]
124             j += 1
125         i += 1
126
127     i = 0
128     while i < N:
129         resid[i] = (rhs[i]*-1)
130         j = 0
131         while j < N:
132             resid[i] = resid[i] + a[i+j*LDA] * b[j]
133             j += 1
134         i += 1
135
136     resid_max = 0.0
137     i = 0
138     while i < N:
139         resid_max = r8_max ( resid_max, r8_abs ( resid[i] ) )
140         i += 1
141
142     b_max = 0.0
```

```

143  i = 0
144  while i < N:
145      b_max = r8_max ( b_max, r8_abs ( b[i] ) )
146      i += 1
147
148  eps = r8_epsilon ( )
149
150  residn = resid_max / (float(N)) / a_max / b_max / eps
151
152  times[2] = total
153  if ( 0.0 < total ):
154      times[3] = ops / ( 1.0E+06 * total )
155  else:
156      times[3] = -1.0
157
158  times[4] = 2.0 / times[3]
159  times[5] = total / cray
160
161  native printf ( "\n" )
162  native printf ( "      Norm. Resid      Resid      MACHEP
163      X[1]      X[N]\n" )
164  native printf ( "\n" )
165  native printf ( " %14f %14f %14e %14f %14f\n", residn,
166      resid_max, eps, b[0], b[N-1] )
167  native printf ( "\n" )
168  native printf ( "      Factor      Solve      Total      MFLOPS
169      Unit      Cray-Ratio\n" )
170  native printf ( "\n" )
171  native printf ( " %9f %9f %9f %9f %9f %9f\n", times[0],
172      times[1], times[2], times[3], times[4], times[5] )
173
174  #
175  #  Terminate.
176  #
177  print ( "\n", "LINPACK_BENCH\n" )
178  print ( "  Normal end of execution.\n\n" )
179
180  timestamp ( )
181
182  return 0

```

181 `main()`

Listing B.3: LINPACK ePython benchmark main file

```

1 external r8_epsilon() [locals:1]->float
2 external r8_abs(float) [locals:1]->float
3 external r8_random(int*) [locals:12]->float
4 external r8mat_gen(float*,int,int) [locals:3]
5 external r8_max(float,float) [locals:1]->float
6 external daxpy(int,float,float*,int, float*,int) [locals:4]
7 external ddot(int,float*,int,float*,int) [locals:5]->float
8 external dscal(int,float,float*,int) [locals:3]
9 external idamax(int,float*,int) [locals:4]->int
10 external dgefa(float*,int,int,int*) [locals:5]->int
11 external dgesl(float*,int,int,int*,float*,int) [locals:3]

```

Listing B.4: LINPACK ePython benchmark externals file

```

1 *****
2 def r8_epsilon():
3 *****
4 #
5 # Purpose:
6 #
7 # R8_EPSILON returns the R8 round off unit.
8 #
9 # Discussion:
10 #
11 # R8_EPSILON is a number R which is a power of 2 with the
    property that,
12 # to the precision of the computer's arithmetic,
13 # 1 < 1 + R
14 # but
15 # 1 = ( 1 + R / 2 )
16 #
17 # Licensing:
18 #
19 # This code is distributed under the GNU LGPL license.
20 #
21 # Modified:
22 #

```

```

23 #    01 September 2012
24 #
25 #   Author:
26 #
27 #    John Burkardt
28 #
29 #   Parameters:
30 #
31 #    Output, double R8_EPSILON, the R8 round-off unit.
32 #
33
34 value = float("2.220446049250313E-016")
35
36 return value
37
38
39 # NOTE: Place function calls here to create the typed variants you
    need
40
41 # Default for linpack_bench:
42 # r8_epsilon () -> float
43 r = r8_epsilon()

```

Listing B.5: LINPACK ePython benchmark r8_epsilon file

```

1  #*****
2  def r8_abs(x):
3  #*****
4  #
5  #   Purpose:
6  #
7  #    R8_ABS returns the absolute value of a R8.
8  #
9  #   Modified:
10 #
11 #    02 April 2005
12 #
13 #   Author:
14 #
15 #    John Burkardt
16 #
17 #   Parameters:

```

```

18 #
19 #   Input, double X, the quantity whose absolute value is desired.
20 #
21 #   Output, double R8_ABS, the absolute value of X.
22 #
23
24 value = 0.0
25 if ( 0.0 <= x ):
26     value = x
27 else:
28     value = (x*-1)
29
30 return value
31
32
33 # NOTE: Place function calls here to create the typed variants you
    need
34
35 # Default for linpack_bench:
36 # r8_abs ( float x ) -> float
37 r = r8_abs(0.0)

```

Listing B.6: LINPACK ePython benchmark r8_abs file

```

1 *****
2 def r8_random(iseed):
3 *****
4 #
5 #   Purpose:
6 #
7 #   R8_RANDOM returns a uniformly distributed random number between
    0 and 1.
8 #
9 #   Discussion:
10 #
11 #   This routine uses a multiplicative congruential method with
    modulus
12 #   2**48 and multiplier 33952834046453 (see G.S.Fishman,
13 #   'Multiplicative congruential random number generators with
    modulus
14 #   2**b: an exhaustive analysis for b = 32 and a partial analysis
    for

```

```

15 #   b = 48', Math. Comp. 189, pp 331-344, 1990).
16 #
17 #   48-bit integers are stored in 4 integer array elements with 12
    bits
18 #   per element. Hence the routine is portable across machines with
19 #   integers of 32 bits or more.
20 #
21 #   Parameters:
22 #
23 #   Input/output, integer ISEED(4).
24 #   On entry, the seed of the random number generator; the array
25 #   elements must be between 0 and 4095, and ISEED(4) must be odd.
26 #   On exit, the seed is updated.
27 #
28 #   Output, double R8_RANDOM, the next pseudorandom number.
29 #
30 #
31 ipw2 = 4096
32 it1 = 0
33 it2 = 0
34 it3 = 0
35 it4 = 0
36 m1 = 494
37 m2 = 322
38 m3 = 2508
39 m4 = 2549
40 one = 1.0
41 r = 1.0 / 4096.0
42 value = 0.0
43 #
44 #   Multiply the seed by the multiplier modulo 2**48.
45 #
46 it4 = iseed[3] * m4
47 it3 = it4 / ipw2
48 it4 = it4 - ipw2 * it3
49 it3 = it3 + iseed[2] * m4 + iseed[3] * m3
50 it2 = it3 / ipw2
51 it3 = it3 - ipw2 * it2
52 it2 = it2 + iseed[1] * m4 + iseed[2] * m3 + iseed[3] * m2
53 it1 = it2 / ipw2
54 it2 = it2 - ipw2 * it1
55 it1 = it1 + iseed[0] * m4 + iseed[1] * m3 + iseed[2] * m2 + iseed

```

```

    [3] * m1
56  it1 = (it1 % ipw2 )
57  #
58  # Return updated seed
59  #
60  iseed[0] = it1
61  iseed[1] = it2
62  iseed[2] = it3
63  iseed[3] = it4
64  #
65  # Convert 48-bit integer to a real number in the interval (0,1)
66  #
67  value = r * ( ( it1 ) + r * ( ( it2 ) + r * ( ( it3 ) + r * ( (
    it4 ) ) ) ) ) )
68
69  return value
70
71  # NOTE: Place function calls here to create the typed variants you
    need
72
73  # Default for linpack_bench:
74  # r8_random ( int iseed[4] ) -> float
75  d = r8_random([0]*4)

```

Listing B.7: LINPACK ePython benchmark r8_random file

```

1  external r8_epsilon()->float
2  external r8_abs(float) [locals:1]->float
3  external r8_random(int*) [locals:12]->float
4
5  #*****
6  def r8mat_gen(a,lda,n):
7  #*****
8  #
9  # Purpose:
10 #
11 # R8MAT_GEN generates a random R8MAT.
12 #
13 # Modified:
14 #
15 # 06 June 2005
16 #

```

```

17 # Parameters:
18 #
19 #     Input, integer LDA, the leading dimension of the matrix.
20 #
21 #     Input, integer N, the order of the matrix.
22 #
23 #     Output, double R8MAT_GEN[LDA*N], the N by N matrix.
24 #
25     init = [ 1, 2, 3, 1325 ]
26
27     j = 1
28     while j <= n:
29         i = 1
30         while i <= n:
31             a[i-1+(j-1)*lda] = r8_random ( init ) - 0.5
32             i += 1
33         j += 1
34
35     return
36
37
38 # NOTE: Place function calls here to create the typed variants you
39 #       need
40
41 # Default for linpack_bench:
42 # r8mat_gen ( float*, int lda, int n )
43 arr = [0.0] * 4
44 r8mat_gen(arr,0,0)

```

Listing B.8: LINPACK ePython benchmark r8mat_gen file

```

1 #*****
2 def r8_max(x,y):
3 #*****
4 #
5 # Purpose:
6 #
7 #     R8_MAX returns the maximum of two R8's.
8 #
9 # Modified:
10 #
11 #     18 August 2004

```

```

12 #
13 #   Author:
14 #
15 #   John Burkardt
16 #
17 #   Parameters:
18 #
19 #       Input, double X, Y, the quantities to compare.
20 #
21 #       Output, double R8_MAX, the maximum of X and Y.
22 #
23
24 if ( y < x ):
25     return x
26 else:
27     return y
28
29
30 # NOTE: Place function calls here to create the typed variants you
    need
31
32 # Default for linpack_bench:
33 # r8_max ( float x, float y ) -> float
34 r = r8_max(0.0,0.0)

```

Listing B.9: LINPACK ePython benchmark r8_max file

```

1 *****
2 def daxpy(n,da,dx,incx,dy,incy) :
3 *****
4 #
5 #   Purpose:
6 #
7 #       DAXPY computes constant times a vector plus a vector.
8 #
9 #   Discussion:
10 #
11 #       This routine uses unrolled loops for increments equal to one.
12 #
13 #   Modified:
14 #
15 #       30 March 2007

```

```
16 #
17 #   Author:
18 #
19 #   FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
20 #   Pete Stewart.
21 #   C version by John Burkardt
22 #
23 #   Reference:
24 #
25 #   Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
26 #   LINPACK User's Guide,
27 #   SIAM, 1979.
28 #
29 #   Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
30 #   Basic Linear Algebra Subprograms for Fortran Usage,
31 #   Algorithm 539,
32 #   ACM Transactions on Mathematical Software,
33 #   Volume 5, Number 3, September 1979, pages 308-323.
34 #
35 #   Parameters:
36 #
37 #   Input, int N, the number of elements in DX and DY.
38 #
39 #   Input, double DA, the multiplier of DX.
40 #
41 #   Input, double DX[*], the first vector.
42 #
43 #   Input, int INCX, the increment between successive entries of DX
44 #
45 #   Input/output, double DY[*], the second vector.
46 #   On output, DY[*] has been replaced by DY[*] + DA * DX[*].
47 #
48 #   Input, int INCY, the increment between successive entries of DY
49 #
50 #
51 #   if ( n <= 0 ):
52 #       return
53 #
54 #   if ( da == 0.0 ):
55 #       return
56 #
57 #   Code for unequal increments or equal increments
```

```

57 # not equal to 1.
58 #
59 if ( (incx != 1) or (incy != 1) ):
60     if ( 0 <= incx ):
61         ix = 0
62     else:
63         ix = ( (n*-1) + 1 ) * incx
64
65     if ( 0 <= incy ):
66         iy = 0
67     else:
68         iy = ( (n*-1) + 1 ) * incy
69
70     i = 0
71     while i < N:
72         dy[iy] = dy[iy] + da * dx[ix]
73         ix = ix + incx
74         iy = iy + incy
75         i += 1
76
77 # Code for both increments equal to 1.
78 #
79 else:
80     m = n % 4
81
82     i = 0
83     while i < m:
84         dy[i] = dy[i] + da * dx[i]
85         i += 1
86
87     i = m
88     while i < n:
89         dy[i ] = dy[i ] + da * dx[i ]
90         dy[i+1] = dy[i+1] + da * dx[i+1]
91         dy[i+2] = dy[i+2] + da * dx[i+2]
92         dy[i+3] = dy[i+3] + da * dx[i+3]
93         i += 4
94
95     return
96
97 # NOTE: Place function calls here to create the typed variants you
    need

```

```

98
99 # Default for linpack_bench:
100 # daxpy ( int n, float da, float* dx, int incx, float* dy, int incy
    )
101 daxpy(10,0.0,[0.0]*10,1,[0.0]*10,1)

```

Listing B.10: LINPACK ePython benchmark daxpy file

```

1 #*****
2 def ddot(n,dx,incx,dy,incy):
3 #*****
4 #
5 #   Purpose:
6 #
7 #       DDOT forms the dot product of two vectors.
8 #
9 #   Discussion:
10 #
11 #       This routine uses unrolled loops for increments equal to one.
12 #
13 #   Modified:
14 #
15 #       30 March 2007
16 #
17 #   Author:
18 #
19 #       FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
    Pete Stewart.
20 #       C version by John Burkardt
21 #
22 #   Reference:
23 #
24 #       Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
25 #       LINPACK User's Guide,
26 #       SIAM, 1979.
27 #
28 #       Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
29 #       Basic Linear Algebra Subprograms for Fortran Usage,
30 #       Algorithm 539,
31 #       ACM Transactions on Mathematical Software,
32 #       Volume 5, Number 3, September 1979, pages 308-323.
33 #

```

```

34 # Parameters:
35 #
36 #   Input, int N, the number of entries in the vectors.
37 #
38 #   Input, double DX[*], the first vector.
39 #
40 #   Input, int INCX, the increment between successive entries in DX
41 #
42 #   Input, double DY[*], the second vector.
43 #
44 #   Input, int INCY, the increment between successive entries in DY
45 #
46 #   Output, double DDOT, the sum of the product of the
47 #       corresponding
48 #       entries of DX and DY.
49 #
50 dtemp = 0.0
51
52 if ( n <= 0 ):
53     return dtemp
54 #
55 # Code for unequal increments or equal increments
56 # not equal to 1.
57 #
58
59 if ( incx != 1 or incy != 1 ):
60     if ( 0 <= incx ):
61         ix = 0
62     else:
63         ix = ( (n*-1) + 1 ) * incx
64
65     if ( 0 <= incy ):
66         iy = 0
67     else:
68         iy = ( (n*-1) + 1 ) * incy
69
70 i = 0
71 while i < n:
72     dtemp = dtemp + dx[ix] * dy[iy]
73     ix = ix + incx
74     iy = iy + incy

```

```

75     i += 1
76
77 #
78 # Code for both increments equal to 1.
79 #
80 else:
81     m = n % 5
82
83     i = 0
84     while i < m:
85         dtemp = dtemp + dx[i] * dy[i]
86         i += 1
87
88     i = m
89     while i < n:
90         dtemp = dtemp + dx[i ] * dy[i ] + dx[i+1] * dy[i+1] + dx[i
+2] * dy[i+2] + dx[i+3] * dy[i+3] + dx[i+4] * dy[i+4]
91         i += 5
92
93     return dtemp
94
95
96 # NOTE: Place function calls here to create the typed variants you
    need
97
98 # Default for linpack_bench:
99 # ddot ( int n, float* dx, int incx, float* dy, int incy ) -> float
100 r = ddot (0, [0.0]*4, 0, [0.0]*4, 0)

```

Listing B.11: LINPACK ePython benchmark ddot file

```

1  #*****
2  def dscal (n,sa,x,incx):
3  #*****
4  #
5  # Purpose:
6  #
7  # DSCAL scales a vector by a constant.
8  #
9  # Modified:
10 #
11 # 30 March 2007

```

```
12 #
13 #   Author:
14 #
15 #   FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
16 #   Pete Stewart.
17 #   C version by John Burkardt
18 #
19 #   Reference:
20 #
21 #   Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
22 #   LINPACK User's Guide,
23 #   SIAM, 1979.
24 #
25 #   Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
26 #   Basic Linear Algebra Subprograms for Fortran Usage,
27 #   Algorithm 539,
28 #   ACM Transactions on Mathematical Software,
29 #   Volume 5, Number 3, September 1979, pages 308-323.
30 #
31 #   Parameters:
32 #
33 #   Input, int N, the number of entries in the vector.
34 #
35 #   Input, double SA, the multiplier.
36 #
37 #   Input/output, double X[*], the vector to be scaled.
38 #
39 #   Input, int INCX, the increment between successive entries of X.
40 #
41 if ( n <= 0 ):
42     return
43 elif ( incx == 1 ):
44     m = n % 5
45
46     i = 0
47     while i < m:
48         x[i] = sa * x[i]
49         i += 1
50
51     i = m
52     while i < n:
53         x[i] = sa * x[i]
```

```

53         x[i+1] = sa * x[i+1]
54         x[i+2] = sa * x[i+2]
55         x[i+3] = sa * x[i+3]
56         x[i+4] = sa * x[i+4]
57         i += 5
58     else:
59         if ( 0 <= incx ):
60             ix = 0
61         else:
62             ix = ( (n*-1) + 1 ) * incx
63
64         i = 0
65         while i < n:
66             x[ix] = sa * x[ix]
67             ix = ix + incx
68             i += 1
69
70     return
71
72
73 # NOTE: Place function calls here to create the typed variants you
74 # need
75
76 # Default for linpack_bench:
77 # dscal ( int n, float sa, float* x, int incx )
78 dscal(0,0.0,[0.0]*4,0)

```

Listing B.12: LINPACK ePython benchmark dscal file

```

1 external r8_epsilon()->float
2 external r8_abs(float)[locals:1]->float
3
4 #*****
5 def idamax(n,dx,incx):
6 #*****
7 #
8 # Purpose:
9 #
10 # IDAMAX finds the index of the vector element of maximum
11 # absolute value.
12 #
13 # Discussion:

```

```
13 #
14 #   WARNING: This index is a 1-based index, not a 0-based index!
15 #
16 #   Modified:
17 #
18 #   30 March 2007
19 #
20 #   Author:
21 #
22 #   FORTRAN77 original by Jack Dongarra, Cleve Moler, Jim Bunch,
23 #   Pete Stewart.
24 #
25 #   C version by John Burkardt
26 #
27 #   Reference:
28 #
29 #   Jack Dongarra, Cleve Moler, Jim Bunch, Pete Stewart,
30 #   LINPACK User's Guide,
31 #   SIAM, 1979.
32 #
33 #   Charles Lawson, Richard Hanson, David Kincaid, Fred Krogh,
34 #   Basic Linear Algebra Subprograms for Fortran Usage,
35 #   Algorithm 539,
36 #   ACM Transactions on Mathematical Software,
37 #   Volume 5, Number 3, September 1979, pages 308-323.
38 #
39 #   Parameters:
40 #
41 #   Input, int N, the number of entries in the vector.
42 #
43 #   Input, double X[*], the vector to be examined.
44 #
45 #   Input, int INCX, the increment between successive entries of SX
46 #
47 #   Output, int IDAMAX, the index of the element of maximum
48 #   absolute value.
49 #
50 value = 0
51
52 if ( n < 1 or incx <= 0 ):
53     return value
54
55 value = 1
```

```

54
55     if ( n == 1 ):
56         return value
57
58     if ( incx == 1 ):
59         dmax = r8_abs ( dx[0] )
60
61         i = 1
62         while i < n:
63             if ( dmax < r8_abs ( dx[i] ) ):
64                 value = i + 1
65                 dmax = r8_abs ( dx[i] )
66             i += 1
67     else:
68         ix = 0
69         dmax = r8_abs ( dx[0] )
70         ix = ix + incx
71
72         i = 1
73         while i < n:
74             if ( dmax < r8_abs ( dx[ix] ) ):
75                 value = i + 1
76                 dmax = r8_abs ( dx[ix] )
77
78             ix = ix + incx
79             i += 1
80
81     return value
82
83 # NOTE: Place function calls here to create the typed variants you
84 #       need
85
86 # Default for linpack_bench:
87 # idamax ( int n, float* dx, int incx ) -> int
88 r = idamax(0,[0.0]*4,0)

```

Listing B.13: LINPACK ePython benchmark idamax file

```

1 external r8_epsilon()->float
2 external r8_abs(float) [locals:1]->float
3 external r8_random(int*) [locals:12]->float
4 external r8mat_gen(float*,int,int) [locals:3]

```

```

5 external r8_max(float, float) [locals:1]->float
6 external daxpy(int, float, float*, int, float*, int) [locals:4]
7 external ddot(int, float*, int, float*, int) [locals:5]->float
8 external dscal(int, float, float*, int) [locals:3]
9 external idamax(int, float*, int) [locals:4]->int
10 external dgefa(float*, int, int, int*)->int
11 external dgesl_proxy(float*, int, int, int*, float*, int)
12
13 *****
14 def dgesl(a, lda, n, ipvt, b, job) :
15 *****
16 #
17 # Purpose:
18 #
19 # DGESL solves a real general linear system  $A * X = B$ .
20 #
21 # Discussion:
22 #
23 # DGESL can solve either of the systems  $A * X = B$  or  $A' * X = B$ .
24 #
25 # The system matrix must have been factored by DGECO or DGEFA.
26 #
27 # A division by zero will occur if the input factor contains a
28 # zero on the diagonal. Technically this indicates singularity
29 # but it is often caused by improper arguments or improper
30 # setting of LDA. It will not occur if the subroutines are
31 # called correctly and if DGECO has set  $0.0 < RCOND$ 
32 # or DGEFA has set  $INFO == 0$ .
33 #
34 # Modified:
35 #
36 # 16 May 2005
37 #
38 # Author:
39 #
40 # C version by John Burkardt.
41 #
42 # Reference:
43 #
44 # Jack Dongarra, Cleve Moler, Jim Bunch and Pete Stewart,
45 # LINPACK User's Guide,
46 # SIAM, (Society for Industrial and Applied Mathematics),

```

```

47 # 3600 University City Science Center,
48 # Philadelphia, PA, 19104-2688.
49 # ISBN 0-89871-172-X
50 #
51 # Parameters:
52 #
53 # Input, double A[LDA*N], the output from DGECO or DGEFA.
54 #
55 # Input, int LDA, the leading dimension of A.
56 #
57 # Input, int N, the order of the matrix A.
58 #
59 # Input, int IPVT[N], the pivot vector from DGECO or DGEFA.
60 #
61 # Input/output, double B[N].
62 # On input, the right hand side vector.
63 # On output, the solution vector.
64 #
65 # Input, int JOB.
66 # 0, solve  $A * X = B$ ;
67 # nonzero, solve  $A' * X = B$ .
68 #
69 #
70 # Solve  $A * X = B$ .
71 #
72 if ( job == 0 ):
73     k = 1
74     while k < n:
75         l = ipvt[k-1]
76         t = b[l-1]
77
78         if ( l != k ):
79             b[l-1] = b[k-1]
80             b[k-1] = t
81
82         daxpy ( n-k, t, a+k+(k-1)*lda, 1, b+k, 1 )
83         #b[k:k+(n-k)] = daxpy ( n-k, t, a[(k+(k-1)*lda):(k+(k-1)*lda)
84         +(n-k)], 1, b[k:k+(n-k)], 1 )
85
86         k += 1
87
88     k = n

```

```

88     while k > 0:
89         b[k-1] = b[k-1] / a[k-1+(k-1)*lda]
90         t = (b[k-1]*-1)
91
92         daxpy ( k-1, t, a+0+(k-1)*lda, 1, b, 1 )
93         #b = daxpy ( k-1, t, a[(0+(k-1)*lda):(0+(k-1)*lda)+(k-1)], 1,
94         b, 1 )
95
96         k -= 1
97     #
98     # Solve A' * X = B.
99     #
100    else:
101        k = 1
102        while k <= n:
103            t = ddot ( k-1, a+0+(k-1)*lda, 1, b, 1 )
104            #t = ddot ( k-1, a[(0+(k-1)*lda):(0+(k-1)*lda)+(k-1)], 1, b, 1
105            )
106
107            b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda]
108            k += 1
109
110        k = n - 1
111        while k > 0:
112            b[k-1] = b[k-1] + ddot ( n-k, a+k+(k-1)*lda, 1, b+k, 1 )
113            #b[k-1] = b[k-1] + ddot ( n-k, a[(k+(k-1)*lda):(k+(k-1)*lda)+(
114            n-k)], 1, b+k, 1 )
115
116            l = ipvt[k-1]
117
118            if ( l != k ):
119                t = b[l-1]
120                b[l-1] = b[k-1]
121                b[k-1] = t
122                k -= 1
123
124        return
125
126    # NOTE: Place function calls here to create the typed variants you
127    need

```

```

126 # Default for linpack_bench:
127 # dgesl ( float* a, int lda, int n, int* ipvt, float* b, int job )
128 dgesl([0.0]*4,0,0,[0]*4,[0.0]*4,0)

```

Listing B.14: LINPACK ePython benchmark dgesl file

```

1 external r8_epsilon()->float
2 external r8_abs(float) [locals:1]->float
3 external r8_random(int*) [locals:12]->float
4 external r8mat_gen(float*,int,int) [locals:3]
5 external r8_max(float,float) [locals:1]->float
6 external daxpy(int,float,float*,int, float*,int) [locals:4]
7 external ddot(int,float*,int,float*,int) [locals:5]->float
8 external dscal(int,float,float*,int) [locals:3]
9 external idamax(int,float*,int) [locals:4]->int
10 external dgefa_proxy(float*,int,int,int*)->int
11 external dgesl(float*,int,int,int*,float*,int)
12
13 #*****
14 def dgefa(a,lda,n,ipvt):
15 #*****
16 #
17 #   Purpose:
18 #
19 #       DGEFA factors a real general matrix.
20 #
21 #   Modified:
22 #
23 #       16 May 2005
24 #
25 #   Author:
26 #
27 #       C version by John Burkardt.
28 #
29 #   Reference:
30 #
31 #       Jack Dongarra, Cleve Moler, Jim Bunch and Pete Stewart,
32 #       LINPACK User's Guide,
33 #       SIAM, (Society for Industrial and Applied Mathematics),
34 #       3600 University City Science Center,
35 #       Philadelphia, PA, 19104-2688.
36 #       ISBN 0-89871-172-X

```

```

37 #
38 #   Parameters:
39 #
40 #       Input/output, double A[LDA*N].
41 #       On input, the matrix to be factored.
42 #       On output, an upper triangular matrix and the multipliers used
43 #       to obtain
44 #       it. The factorization can be written  $A=L*U$ , where L is a
45 #       product of
46 #       permutation and unit lower triangular matrices, and U is upper
47 #       triangular.
48 #
49 #       Input, int LDA, the leading dimension of A.
50 #
51 #       Input, int N, the order of the matrix A.
52 #
53 #       Output, int IPV[T[N], the pivot indices.
54 #
55 #       Output, int DGEFA, singularity indicator.
56 #       0, normal value.
57 #       K, if  $U(K,K) == 0$ . This is not an error condition for this
58 #       subroutine,
59 #       but it does indicate that DGESL or DGEDI will divide by zero if
60 #       called.
61 #       Use RCOND in DGEFO for a reliable indication of singularity.
62 #
63 #
64 #
65 #   Gaussian elimination with partial pivoting.
66 #
67 #
68 #   info = 0
69 #   k = 1
70 #
71 #   while k < n:
72 #
73 #       Find L = pivot index.
74 #
75 #       l = idamax ( n-k+1, a+(k-1)+(k-1)*lda, 1 ) + k - 1
76 #
77 #       #l = idamax ( n-k+1, a[ ((k-1)+(k-1)*lda): ((k-1)+(k-1)*lda)+(n-k
78 #       +1)], 1 ) + k - 1

```

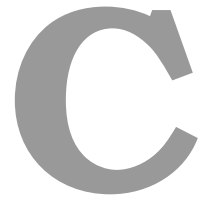
```

73     ipvt[k-1] = 1
74 #
75 # Zero pivot implies this column already triangularized.
76 #
77     if ( a[l-1+(k-1)*lda] == 0.0 ):
78         info = k
79         k += 1
80         continue
81 #
82 # Interchange if necessary.
83 #
84     if ( l != k ):
85         t = a[l-1+(k-1)*lda]
86         a[l-1+(k-1)*lda] = a[k-1+(k-1)*lda]
87         a[k-1+(k-1)*lda] = t
88 #
89 # Compute multipliers.
90 #
91     t = (-1.0 / a[k-1+(k-1)*lda])
92
93     dscal ( n-k, t, a+k+(k-1)*lda, 1 )
94
95     #a[(k+(k-1)*lda):(k+(k-1)*lda)+(n-k)] = dscal ( n-k, t, a[(k+(k-1)*lda):(k+(k-1)*lda)+(n-k)], 1 )
96
97 #
98 # Row elimination with column indexing.
99 #
100    j = k + 1
101    while j <= n:
102        t = a[l-1+(j-1)*lda]
103        if ( l != k ):
104            a[l-1+(j-1)*lda] = a[k-1+(j-1)*lda]
105            a[k-1+(j-1)*lda] = t
106            daxpy ( n-k, t, a+k+(k-1)*lda, 1, a+k+(j-1)*lda, 1 )
107
108            #a[(k+(j-1)*lda):(k+(j-1)*lda)+(n-k)] = daxpy ( n-k, t, a[(k+(k-1)*lda):(k+(k-1)*lda)+(n-k)], 1, a[(k+(j-1)*lda):(k+(j-1)*lda)+(n-k)], 1 )
109
110    j += 1
111    k += 1

```

```
112
113     ipvt[n-1] = n
114
115     if ( a[n-1+(n-1)*lda] == 0.0 ):
116         info = n
117
118     return info
119
120
121 # NOTE: Place function calls here to create the typed variants you
122       need
123
124 # Default for linpack_bench:
125 # dgefa ( float* a, int lda, int n, int* ipvt ) ->int
126 r=dgefa ([0.0]*4,0,0,[0]*4)
```

Listing B.15: LINPACK ePython benchmark dgefa file



Sieve of Eratosthenes code listings

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <string.h>
5 #include <time.h>
6 #include <unistd.h>
7
8 #include "eithne.h"
9 #include "kernels.h"
10
11 #define TARGET_ID 1
12 #define SLEEP_TIME 0000
13
14 int main(void);
15 float cpu_time(void);
16 void timestamp(void);
17 void printarray(const char*, float*, int);
18
19 static int result=0;
20
```

```

21  /* Main data transfer and kernel control function for
22  * Sieve of Eratosthenes benchmark.
23  */
24  int main(void) {
25
26      float t1, t2, time;
27      EithneSharedMem eithne_buffer;
28      static EithneVariable variables[1];
29
30      eithne_buffer = EITHNE_ALLOC_MEM(sizeof(int));
31
32      EITHNE_INIT_HOST(variables, HOST_ID, eithne_buffer +
33          EITHNE_DATA_OFFSET, eithne_buffer);
34
35      /* This will load the code onto the cores */
36      EITHNE_INIT_CORES(1);
37      EITHNE_START_CORES(1);
38
39      usleep(SLEEP_TIME);
40
41      timestamp();
42
43      EITHNE_REGISTER_SCALAR(variables, RESULT, EITHNE_INTEGER, result);
44
45      printf("Sieve of Eratosthenes - %d iterations...\n", ITERATIONS);
46
47      t1 = cpu_time();
48
49      /* The kernel listener will execute the requested function */
50      EITHNE_EXECUTE(TARGET_ID, SIEVE);
51
52      t2 = cpu_time();
53
54      /* Output variables from SIEVE */
55      EITHNE_RECV(variables, TARGET_ID, RESULT);
56
57      time = t2 - t1;
58
59      /* Terminate */
60      EITHNE_FINALISE_CORES;
61
62      printf("There are %u primes less than %u\n", result, SIZE);

```

```
62     printf("%fs\n", time);
63
64     timestamp();
65
66     return 0;
67 }
68
69 /*****
70 float cpu_time ( void )
71 /*****
72 /*
73     Purpose:
74
75     CPU_TIME returns the current reading on the CPU clock.
76
77     Discussion:
78
79     The CPU time measurements available through this routine are
80     often
81     not very accurate.  In some cases, the accuracy is no better
82     than
83     a hundredth of a second.
84
85     Licensing:
86
87     This code is distributed under the GNU LGPL license.
88
89     Modified:
90
91     06 June 2005
92
93     Author:
94
95     John Burkardt
96
97     Parameters:
98
99     Output, float CPU_TIME, the current reading of the CPU clock, in
100     seconds.
101 */
102 {
103     float value;
```

```

101
102     value = ( float ) clock ( )
103             / ( float ) CLOCKS_PER_SEC;
104
105     return value;
106 }
107
108 /*****
109 void timestamp ( void )
110 /*****
111 /*
112     Purpose:
113
114     TIMESTAMP prints the current YMDHMS date as a time stamp.
115
116     Example:
117
118     31 May 2001 09:45:54 AM
119
120     Licensing:
121
122     This code is distributed under the GNU LGPL license.
123
124     Modified:
125
126     24 September 2003
127
128     Author:
129
130     John Burkardt
131
132     Parameters:
133
134     None
135 */
136 {
137     # define TIME_SIZE 40
138
139     static char time_buffer[TIME_SIZE];
140     const struct tm *tm;
141     size_t len;
142     time_t now;

```

```

143
144     now = time ( NULL );
145     tm = localtime ( &now );
146
147     len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p",
148                     tm );
149
150     printf ( "%s\n", time_buffer );
151
152     return;
153 # undef TIME_SIZE
154 }

```

Listing C.1: Sieve benchmark host driver code with Eithne API calls

```

1 #include "eithne.h"
2 #include "kernels.h"
3
4 static EithneVariable variables[1];
5 static char flags[SIZE+1];
6 static int count;
7
8 static void sieve(void);
9
10 /*****
11  /* Initialise the kernel function pointer table
12  *****/
13 void kernel_init(EithneTargetId id, EithneSharedMem buffer) {
14     static EithneKernel kernels[] = { sieve };
15
16     EITHNE_INIT_DEVICE(variables, id, buffer + EITHNE_DATA_OFFSET,
17                         buffer, kernels);
18
19     EITHNE_REGISTER_SCALAR(variables, RESULT, EITHNE_INTEGER, count);
20
21     EITHNE_START_LISTENER;
22 }
23
24 static void sieve(void) {
25     unsigned int i, prime, k, iter;
26
27     for (iter = 1; iter <= ITERATIONS; iter++)
28     {

```

```

28     count = 0;
29     for (i = 0; i <= SIZE; i++) flags[i] = TRUE;
30     for (i = 0; i <= SIZE; i++)
31     {
32         if (flags[i])
33         {
34             prime = i + i + 3;
35             for (k = i + prime; k <= SIZE; k += prime)
36             {
37                 flags[k] = FALSE;
38             }
39             count++;
40         }
41     }
42 }
43 }

```

Listing C.2: Sieve benchmark C kernel code with Eithne API calls

```

1  #ifndef KERNELS_H_
2  #define KERNELS_H_
3
4  # ifdef THREAD
5  #   define SIZE 8190
6  # else
7  #   define SIZE 4095
8  # endif
9  # define ITERATIONS 10000
10
11 enum kernel { SIEVE };
12 enum sharedvar { RESULT=0 };
13
14 extern void kernel_init(EithneTargetId id, EithneSharedMem buffer);
15
16 #endif /* KERNELS_H_ */

```

Listing C.3: Sieve benchmark C kernel include file

```

1 native include(kernels)
2 import time
3
4 SIZE=native const(SIZE)

```

```

5 ITERATIONS=native const(ITERATIONS)
6 flags=[0]*(SIZE+1)
7 i=0
8 prime=0
9 k=0
10 count=0
11 iter=1
12
13 timestamp()
14 print( "Sieve of Erastosthenes - ", ITERATIONS, " iterations...\n")
15
16 t1 = cpu_time()
17
18 while iter<=ITERATIONS:
19     count = 0
20     i=0
21     while i<= SIZE:
22         flags[i] = TRUE
23         i+=1
24     i=0
25     while i<= SIZE:
26         if flags[i] == TRUE:
27             prime = i + i + 3
28             k = i + prime
29             while k <= SIZE:
30                 flags[k] = FALSE
31                 k += prime
32             count+=1
33         i+=1
34     iter+=1
35
36 t2 = cpu_time()
37 time = t2 - t1
38 print("There are ", count, " primes less than 8190\n")
39 print(time, "s\n")
40 timestamp()

```

Listing C.4: ePython Sieve benchmark

```

1 #include "sieve_oly_externs.h"
2
3 #define OLY_ENV_SIZE 16+2
4

```

```

5 void module(Memory *memory) {
6   MKENV(memory, OLY_ENV_SIZE);
7   DECLL("cpu_time", ADDR(0), MKLAMBDA(oly_r$cpu_time$, 1));
8   DECLL("timestamp", ADDR(1), MKLAMBDA(oly_v$timestamp$, 6));
9   DECLI("SIZE", ADDR(2), 8190);
10  DECLI("ITERATIONS", ADDR(3), 10000);
11  DECLA("flags", ADDR(4), NEWARR(sizeof(Int), 1, (LDI(ADDR(2))+1)));
12  Int $init4$[] = { 0 };
13  for (int i=0; i<DIM(ADDR(4), 0); i++) {
14    STAI(ADDR(4), i, $init4$[i%(sizeof($init4$)/sizeof($init4$[0]))]);
15  };
16  DECLI("i", ADDR(5), 0);
17  DECLI("prime", ADDR(6), 0);
18  DECLI("k", ADDR(7), 0);
19  DECLI("count", ADDR(8), 0);
20  DECLI("iter", ADDR(9), 1);
21  (MKBETA(LDL(ADDR(1))), APPLYV(LDL(ADDR(1))));
22  rtl_print(3, 11, "Sieve of Eratosthenes - ", 5, LDI(ADDR(3)), 11, "
    iterations...\n");
23  DECLR("t1", ADDR(10), (MKBETA(LDL(ADDR(0))), APPLYR(LDL(ADDR(0)))));
24  while ((LDI(ADDR(9))<=LDI(ADDR(3)))) {
25    STI(ADDR(8), 0);
26    STI(ADDR(5), 0);
27    while ((LDI(ADDR(5))<=LDI(ADDR(2)))) {
28      STAI(ADDR(4), LDI(ADDR(5)), TRUE);
29      STI(ADDR(5), (LDI(ADDR(5))+1));
30    };
31    STI(ADDR(5), 0);
32    while ((LDI(ADDR(5))<=LDI(ADDR(2)))) {
33      if (((IDXI(ADDR(4), LDI(ADDR(5))))==TRUE)) {
34        STI(ADDR(6), ((LDI(ADDR(5))+LDI(ADDR(5))+3)));
35        STI(ADDR(7), (LDI(ADDR(5))+LDI(ADDR(6))));
36        while ((LDI(ADDR(7))<=LDI(ADDR(2)))) {
37          STAI(ADDR(4), LDI(ADDR(7)), FALSE);
38          STI(ADDR(7), (LDI(ADDR(7))+LDI(ADDR(6))));
39        };
40        STI(ADDR(8), (LDI(ADDR(8))+1));
41      }STI(ADDR(5), (LDI(ADDR(5))+1));
42    };
43    STI(ADDR(9), (LDI(ADDR(9))+1));
44  };
45  DECLR("t2", ADDR(11), (MKBETA(LDL(ADDR(0))), APPLYR(LDL(ADDR(0)))));

```

```

46 DECLR ("time", ADDR(12), (LDR(ADDR(11)) - LDR(ADDR(10)))) ;
47 rtl_print(3, 11, "There are ", 5, LDI(ADDR(8)), 11, " primes less than
    8190\n");
48 rtl_print(2, 7, LDR(ADDR(12)), 11, "s\n");
49 (MKBETA(LDI(ADDR(1))), APPLYV(LDI(ADDR(1)))) ;
50 }

```

Listing C.5: Generated Olympus mnemonic main file for ePython Sieve benchmark

```

1 #include "sieve_oly_externs.h"
2
3 void oly_r$cpu_time$(Memory *memory, Object *self) {
4 RETR((EXPR(Real, clock(), /, Real, CLOCKS_PER_SEC)));
5 }
6
7 void oly_v$timestamp$(Memory *memory, Object *self) {
8 #pragma GCC diagnostic push
9 #pragma GCC diagnostic ignored "-Wformat"
10 #pragma GCC diagnostic push
11 #pragma GCC diagnostic ignored "-Wint-conversion"
12 #pragma GCC diagnostic push
13 #pragma GCC diagnostic ignored "-Wincompatible-pointer-types"
14 DECLI("TIME_SIZE", ADDR(1), 40);
15 DECLI("time_buffer", ADDR(2), malloc(41));
16 DECLI("now", ADDR(3), time(NULL));
17 DECLI("tm", ADDR(4), localtime((Pntr)(ADDR(3))));
18 DECLI("len", ADDR(5), strftime(LDI(ADDR(2)), LDI(ADDR(1)), "%d %B %Y
    %I:%M:%S %p", LDI(ADDR(4))));
19 printf("%s\n", LDI(ADDR(2)));
20 #pragma GCC diagnostic pop
21 #pragma GCC diagnostic pop
22 #pragma GCC diagnostic pop
23 RETV;
24 }

```

Listing C.6: Generated Olympus mnemonic functions file for ePython Sieve benchmark

D

Merlin Engine (VM) code listings

```
1 #define OPCODE_BEGIN(op) op:
2 #define OPCODE_END goto NEXT_OPCODE;
3 #define OPCODE_REGISTRATION
4 #define OPCODE_REGISTRATION_BEGIN OP_REG_BEGIN: \
5     while(1) { \
6         switch((Address)*cp) {
7 #define OPCODE_REGISTRATION_END \
8     } \
9 NEXT_OPCODE: ; } return;
10 #define OPCODE_REGISTER(op) case op ## _OP: goto op; break
11 #define OPCODE_DISPATCH(op) goto *opcodes[ (op) ]
```

Listing D.1: Merlin Engine switch table dispatch macro definitions

```

1 #define OPCODE_BEGIN(op) void op(void) {
2 #define OPCODE_END }
3 #define OPCODE_REGISTRATION register_opcodes()
4 #define OPCODE_REGISTRATION_BEGIN void register_opcodes(void) { \
5     FN_REGISTRATION;
6 #define OPCODE_REGISTRATION_END }
7 #define OPCODE_REGISTER(op) (opcodes[op ## _OP]=(op))
8 #define OPCODE_DISPATCH(op) opcodes[(op)]()

```

Listing D.2: Merlin Engine function pointer dispatch macro definitions

```

1 #define OPCODE_BEGIN(op) op:
2 #define OPCODE_END goto NEXT_OPCODE;
3 #define OPCODE_REGISTRATION
4 #define OPCODE_REGISTRATION_BEGIN OP_REG_BEGIN:
5 #define OPCODE_REGISTRATION_END \
6 NEXT_OPCODE: OPCODE_DISPATCH(*cp); \
7     goto NEXT_OPCODE;
8 #define OPCODE_REGISTER(op) (opcodes[op ## _OP]=&&op)
9 #define OPCODE_DISPATCH(op) goto *opcodes[(op)]

```

Listing D.3: Merlin Engine computed GCC gotos dispatch macro definitions



PicoRV32 support files

```
1 MEMORY {
2     mem : ORIGIN = 0x00000000, LENGTH = 0x0000EC00
3     sdata : ORIGIN = 0x0000EC00, LENGTH = 0x00000800
4 }
5
6 SECTIONS {
7     . = 0x000000;
8     .text : {
9         __init*(.text);
10        trap_vector*(.text);
11        *(.text);
12        KEEP (*(.text))
13        *(*) ;
14    } > mem
15    .sdata : {
16        __global_pointer$ = . + 0x800;
17        *(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata
18        .cst2) *(.srodata .srodata.*)
19        *(.sdata .sdata.* .gnu.linkonce.s.*)
20    } > sdata
```

```
20 _edata = .; PROVIDE (edata = .);
21 end = .;
22 _end = end;
23 }
```

Listing E.1: Custom PicoRV32 linker file

```
1 _init:
2   li x0,0
3   li x1,0
4   li x2,STKPTR
5   li x3,0
6   li x4,0
7   li x5,0
8   li x6,0
9   li x7,0
10  li x8,0
11  li x9,0
12  lw a0,-4(sp)
13  lw a1,-8(sp)
14  li x12,0
15  li x13,0
16  li x14,0
17  li x15,0
18  li x16,0
19  li x17,0
20  li x18,0
21  li x19,0
22  li x20,0
23  li x21,0
24  li x22,0
25  li x23,0
26  li x24,0
27  li x25,0
28  li x26,0
29  li x27,0
30  li x28,0
31  li x29,0
32  li x30,0
33  li x31,0
34  jal main
35  sw a1, -8(sp)
36  sw a0, -4(sp)
```

```
37 end:  
38 ebreak
```

Listing E.2: PicoRV32 register initialisation assembly language file



Olympus mnemonics

```
1 ADDRL(offset)           // Local address at offset
2 ADDRf(level,offset)     // Foreign address at level and offset
3 REF(pntr)               // Dereference pointer
4
5 DECLI(name,addr,value)  // Declare integer
6 DECLR(name,addr,value)  // Declare real
7 DECLB(name,addr,value)  // Declare boolean
8 DECLL(name,addr,value)  // Declare lambda
9 DECLS(name,addr,value)  // Declare string
10 DECLA(name,addr,value)  // Declare array
11 DECLP(name,addr,value)  // Declare pointer
12 DECLC(name,addr,value)  // Declare complex number
13
14 LDI(addr)               // Load integer
15 LDB(addr)               // Load boolean
16 LDR(addr)               // Load real
17 LDC(addr)               // Load complex number
18 LDS(addr)               // Load string
19 LDA(addr)               // Load array
20 LDP(addr)               // Load pointer
```

```

21 LDL(addr)                // Load lambda
22
23 STI(addr,value)           // Store integer
24 STR(addr,value)           // Store real
25 STB(addr,value)           // Store boolean
26 STC(addr,value)           // Store complex number
27 STS(addr,value)           // Store string
28 STA(addr,value)           // Store array
29 STP(addr,value)           // Store pointer
30 STL(addr,value)           // Store lambda
31
32 IDXI(addr,idx)            // Integer array index
33 IDXR(addr,idx)            // Real array index
34 IDXB(addr,idx)            // Boolean array index
35 IDXL(addr,idx)            // Lambda array index
36 IDXs(addr,idx)            // String array index
37 IDXA(addr,idx)            // Array of arrays index
38 IDXP(addr,idx)            // Pointer array index
39 IDXC(addr,idx)            // Complex number array index
40
41 LDAI(addr,idx)            // Load integer array element
42 LDAR(addr,idx)            // Load real array element
43 LDAB(addr,idx)            // Load boolean array element
44 LDAC(addr,idx)            // Load complex number array element
45 LDAS(addr,idx)            // Load string array element
46 LDAA(addr,idx)            // Load array of arrays element
47 LDAP(addr,idx)            // Load pointer array element
48 LDAL(addr,idx)            // Load lambda array element
49
50 STAI(addr,idx,value)      // Store integer array element
51 STAR(addr,idx,value)      // Store real array element
52 STAB(addr,idx,value)      // Store boolean array element
53 STAC(addr,idx,value)      // Store complex number array element
54 STAS(addr,idx,value)      // Store string array element
55 STAA(addr,idx,value)      // Store array of arrays element
56 STAP(addr,idx,value)      // Store pointer array element
57 STAL(addr,idx,value)      // Store lambda array element
58
59 STCR(addr,value)          // Store complex number real value
60 STCI(addr,value)          // Store complex number imaginary value
61
62 MKENV(mem,size)           // Make new environment

```

```

63 MKLAMBDA (code, size)           // Make lambda
64 DYNLDF (name, size)            // Dynamically load function
65 MKBETA (lambda)                // Make beta reduction
66
67 PARAMI (offset, name, value)    // Set integer parameter value
68 PARAMR (offset, name, value)    // Set real parameter value
69 PARAMB (offset, name, value)    // Set boolean parameter value
70 PARAMS (offset, name, value)    // Set string parameter value
71 PARAML (offset, name, value)    // Set lambda parameter value
72 PARAMP (offset, name, value)    // Set pointer parameter value
73 PARAMA (offset, name, value)    // Set array parameter value
74 PARAMC (offset, name, value)    // Set complex number parameter value
75
76 APPLYI (lambda)                // Apply lambda returning integer
77 APPLYR (lambda)                // Apply lambda returning real
78 APPLYB (lambda)                // Apply lambda returning boolean
79 APPLYS (lambda)                // Apply string lambda returning string
80 APPLYL (lambda)                // Apply lambda returning lambda
81 APPLYP (lambda)                // Apply lambda returning pointer
82 APPLYA (lambda)                // Apply lambda returning array
83 APPLYC (lambda)                // Apply lambda returning complex number
84 APPLYV (lambda)                // Apply void lambda
85
86 RETI (val)                     // Return integer from lambda
87 RETR (val)                     // Return real from lambda
88 RETB (val)                     // Return boolean from lambda
89 RETS (val)                     // Return string from lambda
90 RETL (val)                     // Return lambda from lambda
91 RETP (val)                     // Return pointer from lambda
92 RETA (val)                     // Return array from lambda
93 RETC (val)                     // Return complex number from lambda
94 RETV                           // Return from void lambda
95
96 MKARR (addr, size, ...)        // Make array on the stack
97 NEWARR (size, ...)             // Make array in the heap
98 CPYARR (target, src)           // Copy array
99 DIMS (addr)                    // Dimensions of array
100 DIM (addr, dim_no)             // Specific dimension of array
101
102 MKCMLPX (addr, rreal, ireal)    // Make complex number on the stack
103 NEWCMLPX (rreal, ireal)         // Make complex number in the heap
104 CMLPXR (addr)                  // Real part of complex number

```

```
105 CMPLXI (addr)           // Imaginary part of complex number
106
107 NOP                     // No operation
108 INCI (addr)             // Increment integer
109 DECI (addr)             // Decrement integer
110 EXPR (ltype, lhs, op, rtype, rhs) // Expression wrapper
```
