FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

PyPy's Number Crunching Optimization

Just-In-Time Superword Parallelism

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Richard Plangger Bsc
Matrikelnummer 1025637

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 25. März 2015

Richard Plangger   Andreas Krall

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

PyPy's Number Crunching Optimization

Just-In-Time Superword Parallelism

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Richard Plangger Bsc
Registration Number 1025637

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 25th March, 2015

Richard Plangger   Andreas Krall

Erklärung zur Verfassung der Arbeit

Richard Plangger Bsc
Rögergasse 29/31

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

Wien, 25. März 2015

Richard Plangger

Acknowledgements

First and foremost I want express my gratitude to my thesis supervisor Prof. Andreas Krall. Thank you for sharing your knowledge about virtual machines and compiler construction. I enjoyed discussing and tinkering solutions presented in this thesis.

Special thanks to Armin Rigo, Antonio Cuni, Carl Friedrich Bolz and Maciej Fijałkowski for fruitful discussions on computer languages, virtual machines, PyPy internals and other off topic debates. Thank you Maciej for supervising this thesis as Google Summer of Code project.

Thank you Prof. Laurence Tratt for organizing the London sprint. Both the sprint and the Python conference in Bilbao motivated to finish this thesis and continue my work on PyPy.

Above all, I want to thank my partner, Tanja Aigner, for the continuous support throughout my studies as well as my parents for teaching me to pursue my dreams.

Kurzfassung

PyPy ist eine weitbekannte virtuelle Machine für die Programmiersprache Python. Anstatt ausschließlich den Quellcode im Bytecode Format zu interpretieren, stellt PyPy einen Just-In-Time JIT Compiler zur Verfügung.

Außergewöhnlich ist auch die Implementierungssprache RPython der virtuellen Maschine (VM), ein statisch getyptes Subset von Python. Sowohl der Garbage Collector als auch der JIT compiler können zu jeder VM, die in RPython geschreiben ist, generiert werden. Das macht PyPy sowohl zu einer attraktiven Platform für dynamische, funktionale und logikorientierte Sprachen als auch für Instructionset Simulatoren.

In den letzten Jahren wurden Befehlsätze wie SIMD (Single Instruction Multiple Data) in Prozessoren eingebaut. Diese sind nicht nur für multimedia Applikationen nützlich, sondern auch für wissenschaftliche Berechnungen und Simulationen. In der Theorie bieten diese Instruktionen einen Geschwindigkeitsvorteil, der sich linear zur Größe des Vektorregisters verhält. Operationen auf einfachen Fließkommazahlen (32-bit) können dadurch im besten Fall (128-bit große Vector Register) vier mal so schnell ausgeführt werden.

Da Python einfach zu erlernen ist und sich ideal für Datenverarbeitung eignet, wurden nach und nach Bibliotheken gebaut, die wissenschaftliche Berechnungen erleichtern und beschleunigen. NumPy ist ein Beispiel einer weit verwendeten Bibliothek.

Während der Ausführung einer numerischen Berechnung müssen viele Schritte abgearbeitet werden. Dieser interpretative Mehraufwand verlängert die numerische Berechung unnötig. Um diese extra Schritte zu beseitigen wird die Berechnung in einer statisch getypten System-Programmiersprache geschrieben. Das Program wird vor der Ausführung übersetzt und während der Laufzeit der VM aufgerufen.

In dieser Diplomarbeit wird der neue Auto-Vektorisier von PyPy vorgestellt, der Programfragmente automatisch transformiert um SIMD Instruktionen nützen. Die Optimierung ist weder an die NumPy Bibliothek gebunden, noch an eine spezifische Hardwareplattform. Außerdem kann jede VM, die in RPython geschrieben ist, diese Optimierung nützen. Eine empirische Evaulation zeigt wie gut der Vektorisierer auf der x86 Hardwarearchitektur Programgfragmente transformieren kann. Die Ergebnisse zeigen, dass es möglich ist, SIMD Instruktionen auch in einer Dynamischen Sprache zu nützen um Berechnungen zu beschleunigen. Die Komplexität der Transformation ist handhabbar und schnell genug um während der Laufzeit Schleifen zu optimieren.

Abstract

PyPy is a widely known virtual machine for the Python programming language. Opposed to the standard implementation (CPython), it includes a tracing just-in-time (JIT) compiler. The implementation language is a statically typed subset of Python called Restricted Python (RPython). RPython is an abstraction for byte code interpreters and is able to automatically generate a garbage collector and a tracing JIT compiler. Thus it is not only used for PyPy but, many other dynamic/functional/logic oriented interpreters or instruction set simulators.

In the last decade new Single Instruction Multiple Data (SIMD) instruction sets where built into processors to speed up multimedia applications. They are not only useful for multimedia applications but also for scientific applications. The speedup is linearly correlated to the size of the vector register. In theory, given a single precision floating point (32-bit) operation in a loop, in the best case the loop executes the body four times faster using 128-bit vector registers.

Recent developments in scientific computing have drawn attention to libraries (e.g NumPy) for numerical computations. NumPy and others currently remove the interpreter overhead of numerical computations by writing the critical routine in a low level language. They are compiled to the host computers architecture ahead of time. At runtime the language interpreter invokes the foreign function compiled earlier. Since NumPy is a commonly used library, PyPy rewrote parts of NumPy and included it in the standard library. This setup renders most of the critical loops as normal program loops instead of foreign functions.

In this thesis the new auto vectorizer built in to the RPython optimizing backend is presented. It uses only the linear sequence of instructions of a trace to find parallelism. Dependency information is gathered and used to reschedule some of the instructions as vector statements.

This optimization is neither tailored for the NumPy library nor for a specific hardware architecture. Every interpreter written in RPython can benefit from the new optimization. To empirically evaluate the optimization, the x86 assembler backend has been extended to emit SSE4 vector instructions for the optimized traces.

The evaluation shows that it is indeed possible to leverage the speed gain SIMD instruction sets offer and that vectorizing traces at runtime is a feasible optimization technique. This does not only prove to work on NumPy traces, but also for Python loops. The implementation is not very complex and the optimization time is reasonably fast.

Contents

CHAPTER 1

Introduction

1.1 Python Programming Language

Python is a general purpose scripting language. It is well known for its rich tool set of builtin modules, methods and data structures that make it easy to prototype and solve a wide variety of problems. Most of the language features are dynamic and it offloads compile time checks to the runtime. Recent developments allow type information for function parameters, return values and variables to be optionally added[1]. The type hints are used for program verification and runtime checks only.

Not only the parameters, return values or variables are dynamic, but also created objects and their hierarchy. Attributes can be dynamically added or removed. An object's type hierarchy can be exchanged at runtime.

Python adheres to the motto ”Everything is an object“. This allows even functions to be treated as normal objects and forces integer and floating point operations to be boxed within an object. Methods are dynamically looked up and invoked. Arithmetic overflow of a machine word sized integer is silently expanded to an arbitrary precision integer [Cun10].

> *It is a challenging task to optimize dynamic languages to increase execution speed and decrease memory consumption.*

The standard implementation called CPython^{TM} assembles a virtual machine (VM) and the standard library. The VMs execution model interprets byte code instructions and is the most complete implementation [VRD11].

PyPy [Com] is a virtual machine that adds a tracing Just-In-Time (TJIT) compiler.

[1]The Python Enhancement Proposal (PEP) 484 has been recently been accepted and is implemented in Python 3.5. All versions before that did not allow to specify types for function parameters and variables.

In recent years the popularity of dynamic languages such as Python has increased. It is not only an embedded scripting language but very often used as a general purpose programming language. Especially in scientific computing it gained popularity in recent years [Oli07].

Short vector operations are commonly known as SIMD (Single Instruction Multiple Data) operations and benefit scientific computations. Today's processing units, ranging from embedded signal processors to processors in servers, have different vector operations than the ones found in vector computers. Vector computers (often also called super computers) where designed to solve computational expensive tasks that contain a high percentage of data parallelism. [ZC90] Today's processors only provide a SIMD instruction set. SIMD instructions do not operate on vectors containing hundreds or thousands of elements, but on two to sixteen, depending on the type. Even if data parallelism is not omnipresent in an application, SIMD instructions can sometimes still gain speed. [LA00]

1.2 Motivation

PyPy is a Python implementation that speeds up computations and reduces memory usage for Python programs. It is implemented using a subset of Python called "Restricted Python" (RPython).

Opposed to CPython the preferred way removing the interpretative overhead is to use an ahead of time compiler. The critical program fragment is written in a low level programming language. The language interpreter later invokes the highly optimized routine. NumPy is an example of a Python module that is very popular for scientific computing. It adheres the common "paradigm" of writing critical routines in a highly optimized language and is compiled ahead of time. For virtual machines (VM) those "foreign functions" are black boxes and cannot be used to gain any knowledge about the program.

Given the optimizing tracing JIT compiler that is already present in the RPython framework, it is often not necessary to write those programs in a foreign language.

1.3 Problem Statement

PyPy cannot easily use CPython extensions (such as NumPy) that interface with native code. The primary reason for that is the advanced moving garbage collector. It is not able to expose a pointer to foreign code and ensuring that it is not moved or collected.

SIMD instructions are commonly used by ahead of time compilers, but not that often in JIT compilers. With the gaining significance of Python in scientific computing, PyPy cannot yet compete with the AoT compiled NumPy kernels. Thus a solution must be found to use SIMD hardware support.

1.4 Aim of the Work

The goal of this thesis is to enhance the trace optimizer of PyPy. In the current implementation the optimizer in the backend compiler does not try to find candidate loops to vectorize. By definition NumPy arrays are unboxed, homogeneous (every element has the same type) and continuous in memory. The new optimizer will use this opportunity and exploit a SIMD instruction set present on modern RISC/CISC processors. This should lead to enhanced execution speed for arithmetic intensive applications that run on top of PyPy.

The implemented solution has several restrictions it must adhere. First it is not thought of an extension for architectures that provide arbitrary long vector registers, but only SIMD instructions. Second, optimization time should be reasonable fast and at the same time the resulting machine code should be competitive with the numerical kernels of an AoT compilers.

1.5 Methodological approach

The following steps will lead to the desired goal of this work.

1. **Literature Research**. Gather information on JIT compilers, algorithms to analyze loops including loop dependencies, grouping vectorizable instructions, code generation for SIMD and specifics about the PyPy implementation.

2. **Familiarization with PyPy tool chain** and their test driven approach of developing.

3. **Extending the loop analysis** to unroll loops, build a dependency graph and find traces that are suitable to use SIMD instructions.

4. **Filter loop candidates for vectorization early**. Only a subset of the possible loops will be transformed. Some of the filter criteria: Vectorization possible, unrolling factor, usage of continuous & homogeneous arrays.

5. **Implement an algorithm to reveal candidates for SIMD transformation**. This is the most critical part of the implementation.

6. **Adapt the register allocator**. It must recognize the vector registers and correctly spill instructions.

7. **Enhance the assembler to emit SIMD code**. The target instruction set will be SSE.

8. **Evaluate the implementation: Execution speed**. The work will not only contain various micro benchmarks of self written small program snippets, but also evaluate the implementation on a NumPy benchmark suit[2]. It designates itself

[2]`https://github.com/planrich/numpy-benchmarks`, Sept 15

as *"A collection of scientific kernels using the NumPy module for benchmarking purpose"*.

1.6 Contributions to PyPy

The following contributions have been made to the tracing just-in-time compiler backend and it is now able to:

- Create a dependency graph for trace instructions.
- Relax guard instructions to the earliest position in the trace.
- Unroll a trace loop for a factor greater than two.
- Find, extend and combine parallel instructions.
- Schedule a dependency graph and emit vector statements.
- Strengthen guards that protect arithmetic parameters.
- Support accumulation patterns (e.g. sum, all).
- Emit SSE machine code for vector instructions.
- Version trace loops, stitch bridges before the trace is entered the first time.

1.7 Organization of the Work

Chapter 2 presents state of the art techniques to optimize dynamic languages. It covers the most commonly known method based and tracing JIT compilers for dynamic languages. It presents solutions to vectorize loops, including three different techniques to cope with the problem.
Chapter 3 describes PyPy. It briefly touches the translation process from RPython to C source code and in detail describes how the tool chain creates a JIT compiler for any language written in RPython. For the most part it will describe how traces are constructed and introduce the terminology used throughout the document. Furthermore it introduces the internal representation (IR) used in the rest of the document.
The following two chapters 4 and 5 cover the algorithms added to PyPy. It starts of with the classical vectorization approach of AoT compilers and introduces the different dependencies. The rest of Chapter 4 will explain the algorithm in detail. It will cover terminology, complexity, additional preprocessing steps, limitations and other optimizations in Chapter 5 to enhance the generated code. Chapter 6 will show the gain of the current implementation leading to future work and conclusion in Chapter 7.

CHAPTER 2

State of the Art

2.1 Optimizing Virtual Machines

A dynamic typing discipline and object oriented design is appreciated by many programmers, but penalizes the execution performance. The time spent in language constructs that make programs easier to write can be significant. A way to optimize these features is needed and many different solutions have been proposed already.

The absence of type information forces a method based JIT compiler to emit very generic machine code. Nevertheless SMALLTALK and SELF are two well known examples of optimizing method based compilers for dynamic programming languages.

The SMALLTALK optimizing compiler described by Deutsch and Schiffman [DS84] implements an interpreter, a simple translator and a more sophisticated compiler. To handle message passing efficiently the VM uses an inline cache to remember receivers at the call sites. In the native code they represent this by an initially “unlinked” call to the lookup routine. The receiver is then “linked” in a selector/class method cache and the actual method is then re-executed. The simple translator generates exactly four native code bytes for one byte code. A peephole optimization tries to eliminate redundant operations. The sophisticated translator applies several stronger peephole optimizations, keeps the top element in a machine register whenever possible and compiles arithmetic operations inline. This approximately doubles the execution speed, but increases the memory usage. The additionally compiled code takes up to five times more space as the interpreter would.

SELF is also a dynamically typed object oriented language. The VM implementation [CU89] speculatively predicts and infers types of a method and compiles several different method versions. It specializes each version with the known or predicted types. Methods are compiled on demand, keeping track of them in a cache. A cache overflow trashes methods, requiring recompilation upon reentering the method later.

SELFs prototype based object model uses a “clone family” to group objects sharing the same inheritance hierarchy. Many class based object models store the variables and the pointer to the class. This is possible because the object hierarchy cannot change at runtime. SELF takes advantage of an immutable "map" an instance points to. If the object layout is modified a new map is created for this instance. Inline caching considers this "map" and compiles checks into the method ensuring the correctness of the replaced method lookup. Definitions of arithmetic operations are inlined and type information is propagated within SELF expressions. Their so called "customized compilation" is their main contribution. Using their memory representation mentioned earlier and the type information, several different method version coexist in the method cache. The versions are customized by the "map" of the object that receives the message. Their contributions shows a significant gain in execution speed compared to the fastest SMALLTALK implementation.

PSYCO [Rig04] is the predecessor of PYPY and uses JIT type specialization to generate machine code. PSYCO in it's prototype version adds a type specialization technique that queries information about types in functions and assembles small pieces of machine code. Comparing it to tracing, it generates machine code until the first guarding instruction g_n, and later continues to generate yet another assembler piece from g_n to g_{n+1}. Upon guard failure, the VM passes control back to the type specializer and continues to generate a new compiled version. High memory usage caused by over specialization is the primary reason why the project was abandoned.

TRUFFLE [WW12] forms a framework for AST interpreters incorporating a JIT compiler. It shares goals with PYPY and is one of the most related efforts to increase the speed of dynamic languages. TRUFFLE is layered on top of the Java VM. If the AST of the "guest language" is observed in a stable state (e.i. it is not likely to be rewritten), an automatic partial evaluation step infers types and inlines methods. The compiler framework GRAAL compiles the sub part of the AST to machine code. ZIPPY [WB13] is the equivalent Python implementation to PYPY.

The previous examples gave an overview about the possibilities how to optimize dynamic languages. The technique used in PYPY is not novel but more detail is dedicated to it in the next section.

2.2 Tracing JIT Compilation

The building block for tracing JIT compilation has been introduced in the DYNAMO project [BDB00]. DYNAMO is a transparent optimization system that operates on a binary executable. Interpretation starts to execute the program and observes backward branches in a target address cache. By the time a backward branch threshold of an address has been reached, the interpreter switches to a mode where instructions are recorded at the same time as they are executed. This creates a single-entry, multi-exit linear sequence of instructions called a "trace". At every exiting instructions, the sequence that follows the exit (e.g. conditional branch) can be traced and linked to the originating trace. Identical to the trace selection, trace exits are only traced after a certain threshold of executions. The resulting trace trees are accessible in a global associative data structure. Results show that this approach is able to optimize opportunities that manifest themselves only at runtime.

[GPF06] develops HotPath a Tracing JIT (TJIT) compiler for the JamVM[1]. In a similar fashion JamVMs IR is interpreted and backward branches are recorded. The main distinction to Dynamo is that not native code, but the intermediate representation is traced. The compiler deconstructs the stack of the byte code instructions and builds an SSA based IR. The Code generator and register allocator assembles the trace trees in a backward manner. A trace exit (hereafter called guard or side exit) resumes VM execution or directly enters an already compiled trace. To exit the virtual machine all live contents in registers are stored back to the stack frame of the interpreter. When control flow directly leads into a child trace of the parent, it is said that the trace is "stitched" to the parent. Each trace is optimized individually.

[GES+09] can be seen as a successor of the work done in [GPF06] specifically targeting the dynamic nature of the JavaScript language. Notwithstanding the fact that no types are specified by the user code, at trace recording time this information is available. Traces are guarded by type information and much like a function prototype can only be entered when the types match. Function calls can transparently be recorded removing the minimizing the overhead associated with dynamic method lookup.

Whenever a type is unknown (e.g. has been read from a field of an object), in general it must be assumed that the type can change during iterations. A guarding instructions checking the type of the loaded field is inserted ensuring the correctness of the trace execution. Thinking back to Self or Smalltalk an inline cache was used to mitigate the effect of method lookup. Trace-Monkey only needs to record the receiver of the message lookup and insert a guard instruction to side exit the trace when the object hierarchy has changed. Similar to HotPath the register allocation and code generation process traces backwards.

Traces are stitched to the trace trunk instead of recompiling the whole trace tree. The document proposes this extension for future work and multi core architectures that are able to move the compilation into a separate thread.

Tamarin [CSR+09] is a VM implementing both the language ActionScript 3 and a flavor of ECMAScript. Alongside their "conventional" compiler, they added Tamarin-Tracing a tracing JIT compiler for their intermediate byte code. The overall design and heuristics resemble Trace-Monkey but providing the means to both recompile trace trees or only stitch traces. "Tail duplication" increases the emitted machine code. Tamarin-Tracing merges traces on side exits after the third trace has been generated from the initial loop trace. Their implementation is compared, and shows significant speedup to their own previous "conventional" compiler.

SPUR [BBF+10] is a TJIT for CLI. The document tried to find evidence that compiling JavaScript to a typed intermediate language gives the same execution speed as directly tracing and compiling JavaScript. The first translation step inserts profile counters. Overflowing the counter triggers tracing. SPUR dedicates a cache object to each call site that saves the last known receiver of the message. Failing the guard

[1] A Java Virtual Machine. `http://jamvm.sf.net`, July. 2015

updates the cache and stores the new receiver. Guard strengthening is considered a new technique that was primarily introduced in this project. The optimization applied to traces elaborately range from guard implication to load/store elimination/delay, code motion for invariant variables/guards to loop unrolling and constant folding/propagation. All other optimizations the CIL compiler “Bartok” implements are applied too.

[IHWN11] retrofitted a TJIT compiler into the Java Virtual Machine (JVM) J9. The key insight was: Programs with flat profiles cannot be optimized well in a method based JIT compiler. Method inlining would exceed memory limits. The TJIT compiler’s peak performance could reach the speed of the base line compiler. From a technical point of view all techniques used in this paper have already been mentioned. Examples are a hot threshold for loop headers, reuse of the optimization techniques offered by the method base compiler and polymorphic inline caching.

Other tracing VMs for Java include YETI [ZBS07] and Maxpath [BCW+10]. [Pal09] for example implements a TJIT for Lua.

Language Composition, as the name suggests, tries to execute two languages in the same environment. The editor Eco [DT14] allows the mixing of two languages in a convenient way. PyHyp [BDT15] is a virtual machine based on PyPy that is able to execute both Python and PHP programs provided by the Eco editor. The level of composition is fine grained. It allows to intermix PHP and Python expressions in the same source file and even provide the results of a Python expression to a PHP expression as if they where the same language. It provides an interesting solution for code migration. Big projects that have not changed since many years could be step by step migrated to the new language. There is no need for a big bang integration.

2.3 Vectorization Approaches

Vectorization is a well established technique to transform multiple scalar operations into one vector operation. The first operates on a single pair of operands. The latter operation still executes the same function but on multiple pairs at once.

Although vectorization is a subset of concurrent execution it is not the topic of this document and thus omitted in the related work. As a side note this is a complicated problem for the Python programming language. By design both the extreme dynamism and reflective capabilities restrict CPython to not allow concurrent threads execute at once. The problem CPython suffers from is mainly the GIL (Global Interpreter Lock) that both protects the reference counting mechanism and concurrent object modifications. A solution to this problem has recently been proposed and implemented in PyPy STM (Software Transactional Memory). It is a heavy weight solution to let threads be able to run on different CPU cores and at the same time let the runtime handle correct synchronization.

Traditional vectorization as described in [ZC90] analyzes and transforms nested loops. To successfully transform loops the following transformations are applied:

1. **Loop normalization**. Converts the loop header to always start at a common base

(e.g. $i = 0$), increment by one and end at $n - 1$.

2. **Loop collapsing**. Transforms loop nests of level k to a loop nest of level $l < k$.
3. **Loop distribution/fission**. A loop is split into multiple loops over the same range, splitting or extracting statements from the loop body.
4. **Loop interchange**. Two loops change their lexical position.
5. **Loop peeling**. Several iterations before/after the loop are unrolled and executed before/after the peeled loop is entered/exited.

This leads eventually to the transformation of loop statements into vector statements. Both [AK87] and [ZC90] have laid the foundation for loop transformation into vectorized or parallel form. The invention of super computers has lead the search of "auto vectorizers" that could transform sequential programs into semantically equivalent vector forms. There architectures have a very different instruction set than conventional computing platforms. They additionally have vector instructions that operate on thousands to millions of elements in one instruction. The basic principle in [AK87] and [ZC90] is described in Section 4.1 in greater detail. For the sake of completeness a short introduction is included in the following.

The groundwork to transform loops into a semantically equivalent vector form is the data dependence. Most data dependencies emerge from access to array elements. There are many different testing techniques from simple to very advanced. Examples range from the Banerjee's GCD test [Ban88], the Omega test [Pug91] or the Power test [WT92]. All of them consider array subscripts, build equation system and try to find a solution. To prove dependence a solution must exist, otherwise the statements containing the array subscripts are independent.

The well known GCC tool chain implements vectorization in [Nai04]. It is strongly related to [AK87] and constructs scalar dependencies using def-use chains of the tree SSA form. Array subscripts and their dependencies must also consider pointers. For simple subscript expressions Banerjee's GCD is applied. More complex subscripts use accurate tests such as the Omega test. Strongly Connected Components (SCCs) are built from the data dependency graph and loop distribution separates the loops. A strip mining pass prepares the loop to be transformed into SIMD hardware instructions. The article in 2004 indicates that in this version neither different access patterns (such as even/odd) nor an elaborate alias analysis for pointers is applied. Two years later they present the new capabilities in [NZ06]. It is able to vectorize more loops. It improved support for pointer alignment, pointer accesses, conditional operations, reductions and non uniform strides.

Intels vectorizer [BGGT02] uses the same approach outlined in [AK87]. They have put effort into avoiding pitfalls of their architecture and support accumulation as well as realignment by unpeeling the loop body several times. They employ elaborate dependency checking techniques between memory references. Progressively they use more powerful tests. The final test models the problem as an integer linear programming solver. They support reductions, type conversions, pattern such as min or max, constant and variable expansions as well as static and dynamic alignment checking. The results show stunning performance boosts of the 16-bit integer dot product on one selected hardware platform. On the contrary only two out of fourteen benchmarks show significant speedups (i.e. SPEC2000 benchmark suite).

2.3.1 The Polyhedral Model

Another technique makes use of the "Polyhedral Model" [Bas04][BPCB10]. The polyhedral model is a representation of both sequential and parallel programs (i.e. they form a subset of an imperative language such as FORTRAN or C). Loops are described as iteration vectors $\vec{x} = (i_1, ..., i_n)$ and the iteration domain D. An statement can be describe through a set of linear inequalities restricting the space of D. The polyhedron is defined by these inequalities.

A scattering function $\theta(\vec{x}) = T\vec{x} + \vec{t}$ uses a constant matrix T and a constant vector $\vec{t}$. By modifying the constant factors of the scattering function θ, program transformations can be applied. [TNC+09] describes an approach to auto vectorize nested loops using the polyhedral model. A new function θ_S is added defining the total ordered multidimensional timestamps. The matrix used in θ_S captures the relative ordering, expresses code motion, loop fusion and loop fission and captures pipe lining effects.

2.3.2 SIMD Instruction Set Architecture

At the same time the first SIMD instruction sets found their way into production processors (e.g. VIS in 1994 or the MMX in 1997), [CL97] describes an auto vectorizer split into two phases. A source to source compiler first translates loops into a parallel form. The second phase tackles the following problems: alignment of memory load and stores, mask generation, partial stores, expansion, packing and comparison vector operations. Results on the the overall goal to vectorize scalar loops for the MPEG video decoder is not presented, but delayed for the next version of the optimizer.

A very different technique does not consider loops and statements, but only basic blocks. In [KL00] and [PKH07] an algorithm is proposed that operates on a machine independent IR and generates short SIMD instructions. It combines several structurally equivalent statements in unrolled loop bodies into single short SIMD instructions using a data dependency graph. A dedicated lowering step annotates statements with the number of the combined instructions and the primitive type of the computations. BEG, a tree pattern matching code generator, uses this information to emit SIMD instructions for the UltraSPARC VIS ISA. Alignment is another problem that has to be solved for an architecture like UltraSPARC. Both static analysis and dynamic alignment checks help to enter the correct loop version consisting of either scalar or vector statements. [KL00] compares the classical vectorization approach to the basic block vectorization. The latter shows both to be comparable in terms of speedup and simpler to implement than the classical approach. Similar to that [LA00] analyses basic blocks from unrolled loops going through the following steps: Finding adjacent memory references, extending packed instructions, combination and scheduling.

In the following years [FP02][FKÜ05][WEWZ05][SHC05] further evaluate the gain of SIMD instructions in optimizing compilers.

Research in [FP02] implements a source to source compiler that specifically targets Digital Signal Processing (DSP) transformations. Short vector extensions are used to enhance the speed for algorithms (e.g. fast Fourier transformation). For such problems several different fast algorithm exist. SPIRAL is the code generator to enumerate possible algorithms, only differing in data flow. A performance evaluator then chooses the best one among the enumerated. The key to success is to provide permutation matrices that allow the DSP transformation to access aligned memory. They only consider DSP problems that can be naturally be mapped to SIMD instructions. C compiler macros are used to leverage SSE compiler intrinsics. Their contributions transform the mathematical representation of the problem to C source code and thus claim that such vectorization potential cannot be leveraged by conventional auto vectorizers.

Three years later [FKÜ05] in a similar fashion presents a much more complete auto vectorizer for DSP problems. It includes a detailed description of how to overcome the issue of non adjacent memory access. To successfully enhance the speed of DSP transformations an effective solution for these memory hierachy issues is presented. Their evaluation shows that their optimizer is comparable with hand optimized third party DSP transformation libraries.

[WEWZ05] presents a novel approach of "virtual vectors". A virtual vector consists of a length and type of its elements. Their framework creates virtual vectors out of basic blocks, refines them by doing a loop analysis and in the later phase devirtualize them into hardware registers. They introduce several operations on top of virtual vectors. Examples range from reduction patterns, parallel reduction, element slide window to handling alignment for memory operations. The proposed solution is implemented in IBM's XL production compiler and shows that both micro kernels and real benchmarks gain significant speedup.

Up to this point, handling control flow constructs have not been addressed elaborately. Research done in [SHC05] focuses on constructs and tries to find efficient ways to map control flow to SIMD instruction sets. Unrolled basic blocks convert control flow into data dependency using *if-conversion*. Each instruction uses a predicate that indicates if the instruction should be executed in one specific iteration instance. On a VLIW (Very Large Instruction Word) these operations could directly be mapped to hardware equivalents. Unfortunately such instructions are not present on common SIMD ISAs, thus an "unpredicate" step turns the predicated instructions back to normal control flow. The evaluation section shows that control flow constructs carry superword parallelism and this approach is able to leverage the opportunity.

2.3.3 Aligned memory operations

Some SIMD instruction sets only allow the loading and storing to memory addresses that are aligned. This is a very common case for RISC instruction sets. Although on some architectures this limitation has been removed, the unaligned load/store operations are not as efficient as their aligned counterparts. Effort [PKH03][WEW05][EWO04][PKH07][ASRV07] has been put in finding algorithms to remove or mitigate the performance penalty.

In general there are several possibilities to overcome the alignment constraints. It must be either proven that pointer/array accesses are aligned. In the case it cannot be proven, it is possible to emulate unaligned access in software by reordering vector elements of aligned memory loads. Another approach is to decide pointer alignment at runtime. In the normal case this creates at least two versions of the loop. One aligned, the other either unaligned or a scalar loop.

Research done in [PKH03] tries to prove pointer aligned across function boundaries. Alignment is determined by the least significant bits of both user defined pointers and auxiliary pointers. The intra-procedural part of the algorithm annotates pointers with a set of all possible reminders modulo k. If the set of a pointer simply contains $\{0\}$, then the pointer must be aligned. Dynamic allocation assumes to automatically align the memory modulo the constant value k the operating system defines. Pointer arithmetic adds and removes reminder information to of the annotated sets. At the inter-procedural level, alignment information is propagated from the call site to the method body. This can roughly decide 50% of all alignment decisions statically. Moreover this information can reduce the size of the compiled binary on the evaluated programs up to factor of 4.5.

As mentioned earlier, aligned load/store operations can be followed by several reorder operations to move vector elements to the right positions. [EWO04] and [WEW05] show that it is possible emulate alignment at runtime. In addition to that they in cooperate length conversion (conversion between several primitive data types) into the runtime alignment of their new algorithm. The concept of a stream (i.e. a sequence of consecutive array elements) form the foundation of their contributions. Using a stream offset, they define the constraints of a valid SIMD transformation. Misalignment forces the compiler to insert data reorganization instructions to adhere the previous constraints. Because data reorganization can happen at several positions, three different policies (Zero,Eager,Lazy) control the placing of reorganizations.

[ASRV07] describes the performance impact of unaligned memory references. It gives overview over some architectures and the restrictions they impose on unaligned memory access. Their experiments show that it is critical to prove alignment or to provide unaligned access to memory in the instruction set to get performance gains for SIMD instructions. They conclude that for multimedia codes unaligned memory access are very useful.

2.3.4 Interleaved Data Formats

Stride between data elements is another critical issue. Multimedia often operates on various formats. [NRZ06] presents a transformation scheme that efficiently supports vectorization in presence of interleaved data. The only constraint they impose is a stride that is a power of two. Their modifications to the GCC compiler tool chain track each load/store operation with an index to a group. Utilizing this information it can be determined how to reorganize the loaded data into the correct format and reorder it just before storing it o memory. Benchmark programs show significant speedup and highlight the benefit this reorganization can have for interleaved formats.

[RWP06] presents an algorithm to vectorize loops without restricting the reorganization to a power of two. Program source code is transformed to a generic representation followed by an optimization step minimizing the amount of data permutations. The code generator emits the necessary data reorganizations while assembling the final object code. Approximately 77% of data permutations can be eliminated gaining roughly 68% percent speedup for the benchmark programs on the SSE2 platform.

2.3.5 Just-In-Time Auto Vectorizers

Vectorization as an optimization technique in JIT compilers is seldom. One research project extended the Jikes RVM [ESEMEN09] to automatically vectorize loops. The technique that used an extended tree pattern matcher was not improved by follow up projects. Others [RDN+11] try to find data parallelism on byte code level by annotating information that can later used by the JIT VM. [LCF+07] is another example to annotate the byte code generated to enable the VM to vectorize loops.

An approach that couples an offline preparation stage (as mentioned earlier) and a lightweight online stage is implemented in [NDR+11]. The output of the offline stage is a portable vectorized byte code format. They provide a list of idioms the online stage can easily recognize and efficiently map to the target architecture. Configuration in the offline stage is provided to the online stage consisting of alignment properties, cost model metrics and other convenient parameters. GCC was used to emit the vectorized byte code in the offline stage. The online stage was implemented in the free CLR implementation Mono. The document shows that this approach can be used to successfully speedup various benchmark programs.

CHAPTER 3

A Meta Tracing Just-In-Time Compiler

3.1 Just at the Right Time

Building and running a virtual machine spans over several important points in time. PyPy is **translated** from RPYTHON to a binary executable. After starting the VM it parses the input files and creates byte code instructions. A switch dispatch loop **interprets** the byte codes one by one. Once a program fragment has been identified to be interpreted frequently, the **tracing** mechanism records the instructions. It is followed by an **optimization** phase leading to a **compilation** pass. Completing the assembly, **native code is executed**.

Hence forth this thesis will concern itself with the optimization and compilation of traces and for the most part with the optimizations added to PYPY. The translation time is of minor interest, but an overview is given in Section 3.2.

3.2 Translation Overview

PyPy is written in a proper subset of PYTHON called restricted Python (RPYTHON). This setup makes it easy to interpret the source for testing purposes and after a custom translation process delivers similar performance as an interpreter written in C++ or C. The translation of PYTHON source code to C source code is shown in Figure 3.1.

Translation is done on the live objects in memory instead of “dead” source files. The whole program is imported and initialized by any Python interpreter. After this “bootstrapping” process most of the dynamic features are disabled and analysis of the objects in memory begins.

The annotation pass infers types on the control flow graphs in the object hierarchy. Each variable declaration missing in the original program is annotated with a type. Live

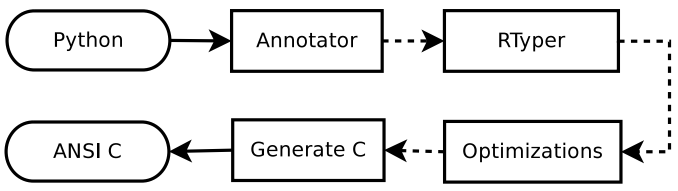


Figure 3.1: RPython transformation step to C source code.

objects have type information attached and it can be used to propagate this information. Every further object construction provides the type and parameter types. If the type inference deduces several types for a variable, the transformation is stopped. Thus PyPy is considered a statically typed language.

Annotation finds the most precise type information for a given variable. As input it takes a set of functions and abstractly interprets their control flow graphs (CFG). A fix point algorithm forwards annotations from the beginning of the entry function into each block of the CFGs. In the presence of a loop, type information might change after several visits of the inner blocks. Each time the more general type is annotated. The algorithm stops by either reaching the most general type, or no changes have been detected.

The result of the previous steps is a statically typed set of functions that are then transformed to a simpler representation in the RTyper pass. Annotated blocks still represent a very high level description of computations needing many builtin functions Python provides. The resulting “simpler” operations can be easily translated to C source code and use types that are known in that language.

An optimization pass transforms the low level instructions. Function inlining, malloc removal, escape analysis and stack allocation are applied. The C source code is written to disk and translated using a conventional C compiler. [AACM07].

3.3 Tracing the Meta Level

To understand how the meta tracer works the mechanics of the interpreter are the bare essentials. Listings 3.1 and 3.2 show the Python 2.7 source code and the disassembled byte code for the `Cobra.bite` method.

```python
class Snake(object):
  def bite(self, other):
    pass # no harm done

class Cobra(Snake):
  def bite(self, other):
    other.poision += 10
```

Listing 3.1: A sample Python object hierarchy

```
0  LOAD_FAST     1 (other)
3  DUP_TOP
4  LOAD_ATTR     0 (poison)
7  LOAD_CONST    1 (10)
10 INPLACE_ADD
11 ROT_TWO
12 STORE_ATTR    0 (poison)
15 LOAD_CONST    0 (None)
18 RETURN_VALUE
```

Listing 3.2: PyPy's byte code invoking Cobra.bite in Listing 3.1

Each byte code listed in Listing 3.2 spans over several RPython statements that modify the Python frame[1]. The following describes how `INPLACE_ADD` is implemented in the PyPy interpreter. First it removes the two top most elements from the stack, searches for the right method to dispatch according to the Method Resolution Order (MRO)[2] and executes the operation. If both types are integer boxes a new box with the summed value as content is created. At the same time it is ensured that the integer is not overflown. The result of the operation is added to the top of the stack. In any other case, the method `__add__` of the class object is invoked.

When the tracing enters the `bite` function, `INPLACE_ADD` and `LOAD_ATTR poison` are the only two instructions that yield entries in the trace history:

```
LOAD_ATTR posion
  inst_map = getfield(other, descr=<Field inst_map>)
  guard_value(inst_map == snake_map) # snake_map is constant
  int_box = getfield(other, descr=<Field inst__value0>)
  guard_nonnull_class(int_box, IntObject)
INPLACE_ADD
  i0 = getfield(int_box, descr=<Field inst_intval>)
  i1 = int_add_ovf(i0, 10)
  guard_no_overflow()
```

`LOAD_ATTR poison` loads the instance map of the object and compares it to a constant pointer value. Similar to Self [CU89] the idea is to share object layout between similar instances [Bol13].

Provided that the layout of the object referenced by the name `other` (this is a instance of `Snake`) does not change the `guard_value` will succeed. The next `getfield` operation loads the actual integer box of the object and ensures that it is indeed an instance of an integer box. The three operations `INPLACE_ADD` ends up with retrieving

[1]Similar to the activation record, a Python frame stores local variables, the byte code itself and more state needed by the interpreter.

[2]MRO is explained at `https://www.python.org/download/releases/2.3/mro/` July 2015

the actual value of integer box (getfield), adds 10 to it and checks if an overflow occurred (guard_no_overflow).

Ultimately (not shown in the listing) a new integer box is created, the contents of i1 are stored and the integer box is saved in the slot of the Snake instance.

Listing 3.3 shows that one byte code forms several different intermediate instructions. They are **not manually specified, but automatically generated** at translation time.

3.4 Interwoven JIT Compilation

In a nutshell the tool chain deduces the tracing information from the control flow graph (CFG) of the RPython program. In the bootstrapping phase the program adjusts certain points in the main interpreter loop. A JitDriver is the portal to switch from the interpreter to the tracer or an already compiled trace. The following listing shows the necessary hints the VM developer must add.

```python
driver = JitDriver(greens=['code','pc'], reds='auto')
def switch_dispatch(ctx,code):
  pc = 0
  while True:
    driver.jit_merge_point(code=code, pc=pc) # jit may exit
    byte_code = code[pc]
    if byte_code == INT_ADD:
      # ...
    elif byte_code == JUMP:
      # ...
      driver.can_enter_jit(code=code, pc=pc) # jit may enter
    pc += 1
```

Using the RPython tool chain outlined in 3.2 the translation process rewrites the driver call sites can_enter_jit and jit_merge_point. The former opens a “portal” to the tracer or enters native code execution. The latter generates exception code to resume interpretation. These are the only annotations that are needed to enable the trace compiler [DC].

All actions of the interpreter are traced, not the user program. This creates an abstraction layer called the “meta level” [Cun10]. To accomplish this the CFG of the interpreter loop is analyzed. Each block within the CFG adds up the actions to be taken by the interpreter. A JitCode persists this information.

As soon as the interpreter actions are traced, the right position within the JitCode must be found. A tailored interpreter for a JitCode executes and records the instructions at the same time. After eventually reaching the loop header (identified by the green key) the trace loop is closed and the recorded history is forwarded to the optimization and compilation backend. If the green key is not reached and a certain amount of instructions has been recorded, the tracing is aborted.

The resulting JIT compiler is complete for the RPython language, modular, reusable and separated from the language semantics. New languages can just reuse the TJIT compiler. In addtion the setup leaves room for the framework and the standard library [DC].

There is one down side to this setup that manifests itself in the size of the resulting shared library that is delivered with the PyPy executable. It is optimized for speed and not for size, thus the condensed JitCodes take up several megabytes of space.

RPython is used frequently. Examples range from Topaz[3] (Ruby), Pycket[4] (Racket), Prolog [BLS10] to Hippy[5] (PHP). All of them are in different states of completion.

3.4.1 Intermediate Representation

The semantics of each individual operation code must not be specified by the programmer, but they are automatically generated by the known RPython semantics. They are condensed into the final executable and describe the tracing intermediate representation. The IR instructions are specified as follows:

$$r = name(a_1, ..., a_n, descr)[f_1, ..., f_n]$$

The IR specification consists of an unique *name* associating the semantics that is required to be implemented in the assembler backend. It is able to return a new variable r and can take several arguments $a_1, ..., a_n$. For instructions such as guards or loading operations, more information is attached in the *descr* field. The descriptor is a polymorphic container to save information that cannot be described using a variable. An example would be type information of a memory load/store operation or additional information on how to side exit a guarding instruction. The special arguments for guards are the "failing arguments" $f_1, ..., f_n$. They represent all the variables that are live at the given position and must be considered when side exiting the trace. Henceforth variables are denoted as "boxes" as well. Boxes have three types: Integers, Pointers or Floating point. The prefixes in the variables are i,p,f respectively.

The following listing shows a trace loop using the IR notation introduced earlier.

```
label(p1,i2,f3)
f4 = load(p1,i2,descr=float64)
f5 = float_mul(f4,f3)
i4 = int_add(i1,8)
guard_true(i2 < 100,descr=resume_at_position) [i4,f5]
jump(p1,i4,f5)
```

Listing 3.3: Trace instructions forming a trace loop

The `label` and `jump` both have a descriptor attached, but it is omitted in the listing. It describes an unique target address (i.e. the label's absolute address). `float64` denotes a type descriptor for the loading instruction and `resume_at_position` is a

[3]Topaz `http://topaz.readthedocs.org/en/latest` July 2015
[4]Pycket `https://github.com/samth/pycket` July 2015
[5]Hippy `http://hippyvm.com/` July 2015

guard resume descriptor, storing enough information to continue in the interpreted mode whenever the condition `i2 < 100` is not satisfied. Note that the comparison is inlined as a parameter. [DC]

Boxes are immutable. This leads to a Static Single Assignment[CFR+89] (SSA) representation for scalar variables. Whenever box content must be modified, a new unique box is created representing the resulting calculation. ϕ-nodes are not explicitly modeled, because there is no control flow join other then the loop header. There are three instruction types:

- **Pure/Elidable**: Free of any "observable" side effect and referential transparent. The operation with the same operators can be replaced by it's result. This does not change the program semantics. E.g. `int_add`, `strlen`.
- **Side effect free**: Operations that do not change the state of the interpreter or any associated data structure. E.g. `load`, `getfield`. Note that the following holds: **Pure** $\subseteq$ **Side effect free**.
- **Guards**. E.g. `guard_true`, `guard_class`.
- **Others**: Any other operation that changes state or does not satisfy the above constraints. E.g. `call`, `label`.

3.4.2 Identifying loop headers

TJITs need a way to identify loop headers. The program counter could be one indication that the same loop header has been reached. This could be a necessary condition for language X but might not fit for Y. Thus the JitDriver gives the language implementer the choice to provide "green variables". The final identification is a hash of all green variables. In a similar fashion the "red variables" are all other boxes used in the loop. Examples for the red variables would be the activation record ("frame object") and the execution context. [DC]

3.4.3 Virtual machine states

During the execution PyPy switches between various states. This separation is inspired by [Cun10].

- **Interpret**. Byte code instructions are dispatched one by one modifying the internal state of the VM. This is the default mode PyPy starts its execution. As shown in Listing 3.4 backwards jumps are instrumented to count the loop iterations. A loop is considered **hot** after exceeding the iteration threshold. The interpreter enters the **Trace** state.
- **Trace**. The next execution of the program loop records the instructions as a history. If the number of traced instructions does not overrun, the sequence is provided as input to the optimizer (State **Optimize**).

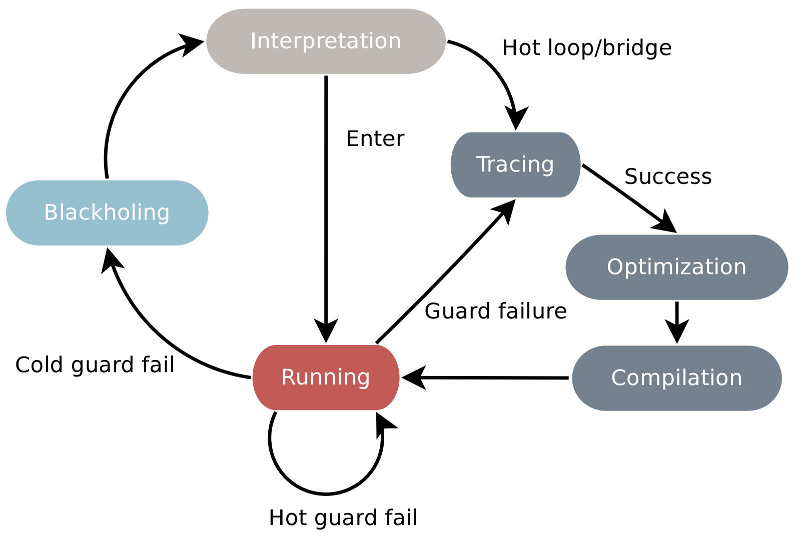


Figure 3.2: All interpreter states the TJIT switches between

- **Optimize**. This is a critical step to simplify the trace, remove and exchange instructions. It applies commonly known optimizations such as constant folding, constant propagation, strength reduction, invariant code motion, partial redundance elimination and many others. Both guard implication and guard strengthening are two TJIT optimizations also implemented in PyPy. This state is immediately followed by **Compile**.

- **Compile**. This step transforms the tracing history to native machine code at runtime. It covers both register allocation and code generation. PyPy offers several different backend implementations such as x86 both 32 and 64 bit, ARM and an incomplete PPC backend. After the compilation succeeded, the interpreter enters the **Run** state.

- **Run**. The VM executes any compiled trace loop. Guard exits record how often instructions have failed eventually tracing the guard exit instructions and stitching them to the guard instruction. Before a new trace is stitched the **Blackhole** interpreter reconstructs the interpretation state.

- **Blackhole**. The state to reconstruct the environment to continue interpretation. This is necessary when exiting a trace at any guarding instruction that has no stitched trace to it. By the time the next byte code can be sanely executed and the normal interpreter resumes.

Figure 3.2 shows the states and various transitions between them. An edge description containing “cold” means that the threshold has not been reached yet. On the contrary “hot” means that the execution can follow a stitched trace or that a candidate for tracing has been found.

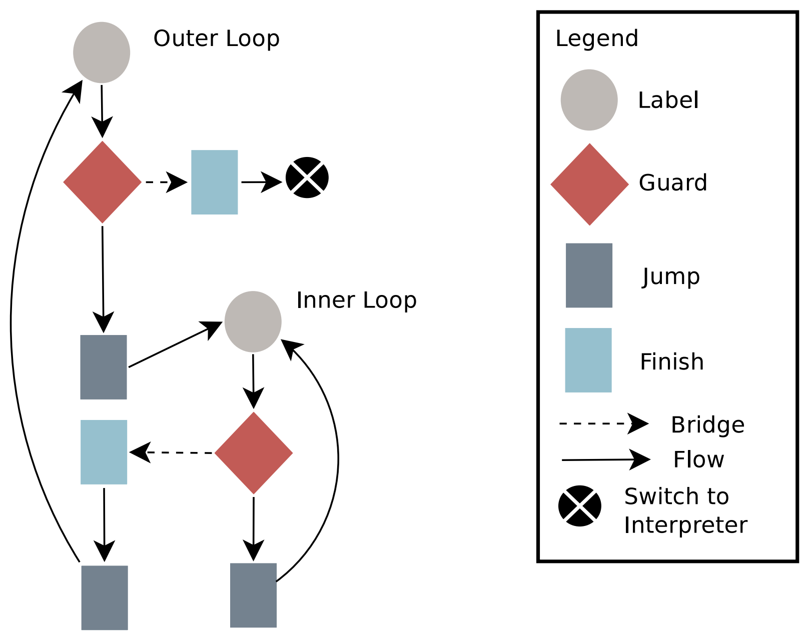


Figure 3.3: A trace tree constructed by RPython's tracer

3.4.4 Building Trace Trees

In the current implementation the traces are stitched to failing guard instructions. Trace trees can be nested with outer trees which makes the tracing mechanism fully recursive. This is shown Figure 3.3. Inner loops of user programs exceed the hotness threshold before their direct parent, thus they are subject of compilation before any parent. PyPy has a special instruction `call_assembler` which enters an inner loop and the instruction `finish` to exit to a parent loop or to the interpreter.

In Figure 3.3 the building of the trace starts at the inner loop. As soon as the threshold exceeds the inner loop is traced. Neither has the outer loop been attached nor the bridge leading into the `finish` instruction. Little by little the outer loop counter and guard counters increment. The latter creates the bridge leading into the `finish` instruction, tracing all instruction up to the jump to the outer loop label and leading back to the `call_assembler` instruction of the inner loop. At this point the trace is optimized and compiled. Eventually the outer loop is exited at the marked guard instruction using the blackhole interpreter or (after sufficiently many exits) compiles a bridge to switch to the interpreter.

3.5 Optimization Framework

The optimizer in the backend currently makes a single pass over the trace loop or bridge and has a slightly more complex optimization called “unrolling”. The following provides an incomplete list of optimizations.

- **Rewrite**. Commonly known as strength reduction including arithmetic and logical simplifications. E.g. An integer multiplied by 2 is equivalent to the bits shifted to the left once (e.g. $x * 2 \equiv x << 1$). Other examples for simplifications would be $x \& 0 \equiv 0, x - 0 \equiv x$.

- **Guard implication**. If a guard logically implies a following guarding instruction, the second can be omitted.

- **Guard strengthing**. If a guard that is at a later position in a trace implies an guard at any earlier position it is possible to replace the first guard with the second guard.

- **Allocation removal**. This is based on partial redundancy elimination [Bol13]. It assumes that some variables to be constant and operations using these variables can be folded away. It includes an escape analysis which optimistically assumes that every object allocation is just used for a few instructions and does not escape. Such objects are also called virtual objects. Even if the object escapes the trace, the guard failure restores the object at the trace exit.

- **Load/Store elimination**. If either several loads or several stores use the same location, it is possible to remove either dead stores or redundant loads. The latter is only allowed if the memory locations in between have not changed.[6]

Unrolling is the PyPy term for loop peeling and invariant code motion done on trace loops. It is one of the optimizations that tries to move operations out of the loop body into the first iteration of the loop [ABF12]. Guard implication and guard strengthening can take full advantage of the first peeled loop body. This is one of the most powerful optimizations in RPython. The kernel to compute e.g. a vector addition in NumPyPy consists of roughly IR 50 instructions when it is first traced. After unrolling has completed it's second pass, the loop body is optimized down to approximately 10 IR instructions.

[6] `http://morepypy.blogspot.it/2010/07/comparing-spur-to-pypy.html` July 2015

CHAPTER 4

Exploiting Superword Parallelism

4.1 Algorithmic Approach

Automatic transformations are omnipresent in modern and optimizing compilers. Vector instruction sets are meant to be used in loops to advance the loop more quickly and at the same time compute more operations. In the following two different approaches are distinguished: "Classical" and "Basic Block". Both of them solve the same problem, but see the problem from a different angle. Array processors execute faster the bigger the arrays. The need for abstraction and finding the maximum of parallel instruction drove the invention of "classical" vectorization.

4.1.1 The "Classical" Approach

The notation used in the following is similar to the one found in [ZC90]. It is called "classical" because most AoT compiler use this approach to generate code for more than a decade (e.g. the GCC compiler tool chain [Nai04]).

In this context of this document we informally define two statements "**dependent**" if their ordering within the program is relevant. Exchanging their position (in the text file, or in the IR) generally yields a different result. "**Independent**" is defined analogous: Two statements are independent if and only if their ordering in the program does not alter the semantics. When we speak of a **dependency** there are three types:

1. **True Dependency** (δ or δ^t). Two statements are dependent by the flow of the program. The output of the first is used as parameter to the second. Synonym for "read-after-write".

2. **Anti Dependency** (δ^a). Present if the first statement reads a variable that is written in the following statement. Synonym for "write-after-read".

```python
a,b,c = ... # arrays
for i in range(30):
  a[i] = b[i] + c[i]
```

Listing 4.1: Loop calculating the element wise sum of two vectors

3. **Output Dependency** (δ^o). Two statements write to the same variable. Synonym for "write-after-write".

These are called data dependencies. For the sake of completeness there is one dependency that must be preserved in presence of conditional branches: Conditional dependencies. Both anti and output dependencies can be eliminated renaming the variables.

Just considering two statements to be dependent is not sufficient. All dependencies must be considered, forming a "**data dependency graph**" (DDG). Each statement is a node and dependencies are represented as directed edges.

Building a DDG for programs (e.g the example in Listing 4.1) is a critical part of vectorization. The ultimate goal is to prove statements to be independent and derive "packed" instructions that cope with the loop within one statement. The same loop in Fortran vector notation is `a(0:30) = b(0:30) + c(0:30)`. An array processor most likely would support this operation in their instruction set. Using a SIMD processor extension, the loop must be strip mined.

4.1.2 Dependency for loop nests

Before dependencies are analyzed the loop nest is transformed to an equivalent form "normalizing" the index. After this simplification each loop's index range starts at the base value (e.i. zero) and increments by one to the upper bound $n - 1$.

Dependencies do not only apply to textual statements, but can be loop "carried". Allowing array subscripts in loop bodies must track the order of some iterations in the loop. An "**iteration instance**" is the execution of the loop body with a fixed index i. For a nested loop an iteration instance is expressed as n tuple of the indices. S_y depends on S_x if an input value used by S_y is produced by S_x. This requires to check iteration instances. In the loop context a statement can even depend on itself. Given the loop in Listing 4.2. S_1 must not depend upon another iteration instance of S_1. If two iteration instances i_1, i_2 $(1 \leq i_1, i_2 < n)$ s.t. $f(i_1) = g(i_1)$ exist, then there is a loop carried dependency.

A dependency exists if a solution to the equation

$$f(i_1) - g(i_2) = 0$$

```
for i in range(1,n):
  a[f(i)] = a[g(i)] # S_1
```

Listing 4.2: A loop copying a range of elements. Inidices are generic functions.

```
for i in range(1,n):
  a[i] = a[i-1]
```

Listing 4.3: Listing 4.2 with the generic functions replaced by indices.

can be found. Whenever either f or g are arbitrary functions, finding a solution becomes complex. If both f and g are linear expressions the problem is tractable. Assuming that both subscript indices of the form:

$$f(i) = ia_0 + a_1$$
$$g(i) = ib_0 + b_1$$

An integer solution exists to the equation $xb_0 - ya_0 = b_1 - a_1$ using the theorem for Diophantine equations. $ax + by = n$ iff $gcd(a, b)|n$ [AK87]. Assuming that $f(i) = i$ and $g(i) = i - 1$ as shown in Listing 4.3. Every iteration (but the first) depends on the previous iteration.

The far most common case in practice yields $n = b_1 - a_1 = 1$ which is too restrictive in practice. Thus Banerjee's [Ban88] equality test dependencies considers a real solution instead of an integer solution in the iteration range.

[AK87] describes the final dependency test as follows: Provided that both f and g are linear, if either $gcd(a_1, b_1)$ does not divide $b_0 - a_0$ or Banerjee's [Ban88] inequality does not hold, the two statements do not depend on each other. It is not an exact test, but solves many practice relevant cases.

Up to now only self dependence has been discussed. It is often more interesting if two statements depend on each other. Figure 4.1 shows both the source code and the dependency graph. To determine the loop carried dependencies each statement must test the array subscripts using the same array. As an example statement S_4 carries a true backwards dependence to S_3. It is called backwards because S_4 δ_b S_3, but S_3 is textually before S_4 in the loop body. S_4 writes to `c[j+1]` and S_3 reads from `c[j]`. Finding a solution to $i_1 + 1 = i_2$ in the iteration domain proves a dependency. A solution[1] to the equation is $i_1 = 1, i_2 = 2$.

There are several more algorithms to test dependencies for statements. The Omega test [Pug91] or the power test [WT92] are two examples that prove the existence of dependency on loop nests.

4.1.3 Vectorizing statements

Given a dependency graph for a loop nest an algorithm is proposed in [ZC90] that calculates the Strongly Connected Components (SCCs). An algorithm that yields SCCs is Trajan's algorithm [Tar72]. This creates an abstraction on the graph grouping together nodes that are in a dependency cycle.

[1]This assumes a big enough iteration domain for both `n` and `m`.

```
for i in range(n):
  for j in range(1,m):
    b[j-1] = d[i]        # S1
    a[i] = b[j-1] * 2    # S2
    c[j-1] = c[j] + 1    # S3
    c[j+1] = a[i]        # S4
```

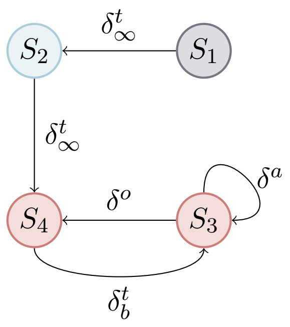


Figure 4.1: An nested loop showing its dependency graph. The resulting SCCs share the same color. δ_∞ denotes dependencies that occur in the loop body (i.e. loop independent dependencies). All others are loop carried. δ_b denotes a backwards dependency.

SCCs are topologically sorted and processed in the new order. Either a new loop is generated whenever the component spans over multiple statements or a vector statement is generated.

Figure 4.1 shows a sample program and their SCCs. There are three SCCs, $C_1 = \{S_1\}, C_2 = \{S_2\}, C_3 = \{S_4, S_5\}$. The resulting loop can fully vectorize statement S_1 and S_2 since they are both in a separate SCC and both of them do not contain a loop carried dependency. S_3 and S_4 are emitted in the same loop nest preserving the lexical ordering.

This might not handle loop nests that contain vectorization potential at higher loop levels. Thus the algorithm is refined in a recursive manner.

A loop nest at level k transforms all statements into vector form (proven that the program semantics are preserved) and removes the loop for the new vector statements. If a deeper level $k + 1$ exists and not all statements have been transformed it recursively invokes the procedure at level $k + 1$. While proceeding to level $k + 1$, all dependencies that exist on level k are removed, further increasing the potential for vectorization.

Additional optimizations such as loop interchange, scalar and constant expansion, index set splitting, loop peeling, node splitting are other transformations that increase the potential for vectorization [ZC90].

4.1.4 Super word parallelism in basic blocks

As described earlier there are several hard problems involved in generating vector statements out of loop nests. The high abstraction level for loop nests and their contained statements make it possible to reason and solve complex problem instances. Obviously this solution provides high degree of abstraction. A trace loop is one specific iteration of a single loop. Although PyPy is able to connect nested trace loops, in the current setup, trace trees are not recompiled but stitched. Furthermore the elimination of guards in the outer loop is limited opposed to the UNROLLING optimization.

The algorithm described by Larsen [LA00] operations on the basic block level of program fragments. At a first glance this seems to perfectly match with the requirements needed for a TJIT because one trace loop is in essence one monolithic basic block. The fundamental idea is that a basic block might contain enough parallelism for this transformation. A well known mechanism to reorder instructions is “scheduling”. By grouping independent operations a scheduler could efficiently prepare the code generation for a SIMD architecture.

A successful transformation often needs more parallel instructions than the user provides in the loop. Simply unrolling the loop body several times can solve this issue.

The rest of this chapter will explain both the general algorithm and cover details of the integration within the RPython optimization backend. The following list introduces terminology used in the next sections. Henceforward if we speak of VecOpt we refer to this implementation.

- **Array/Vector**: A continuous chunk of memory containing primitive or complex objects. In the following only primitive arrays are of interest (numbers and integers are primitive, but not a structured object).

- **Pair**: A group of two statements that are independent. The program semantics is not modified if both of them are executed at the same time. The element at index zero is also called “left” and at index one “right”.

- **Pack**: A group of n statements that are independent ($n \geq 2$).

- **Variable index**: A quadrupel `(v, a, b, c)`. The actual value relative to `v` can be calculated with the expression: number $= v\frac{a}{b} + c$. a,b and c are constants. The quadrupel is often written as `(v * a/b + c)`.

- **Memory reference**: A tuple (array$_i$, variable_index) describing a memory address.

- **Data Dependency Graph** (DDG): A Directed Acyclic Graph (DAG). It is built using a sequence of statements where each statement is a node in the graph and the edges represent one or more of the three data dependencies.

- **Vector register**: A special hardware register of the target platform that is able to be used in packed operations. In the following we assume that a vector register has the same properties as it does in the SSE4.1 specifications. This means that the register is able to store up to 128 bits of either packed floating point or integer numbers.

- **Statement/Instruction/Operation**. All of them denote a syntactical expression that is evaluated in one atomic step. A statement denotes a “user” program step (typically an expression with an assignment). An instruction denominates a step in the IR. Instruction and operation is used as a synonym, but operation can also refer to an arithmetic/logical operator executed on variables.

4.2 Superword parallelism on trace sequences

The optimization routine is outlined in Algorithm 1. It shows the preparation routine for a trace loop and the algorithm to vectorize trace loops. Relax in Algorithm 1 finds guarding operations and moves them to the beginning of the loop. The guard is then called an "early exit". For example the guard protecting the loop index is executed at the end of the trace. This check at the end adds a dependency to the next load instruction and the previous store instruction of the unrolled trace loop. Moving this check to the beginning of the loop, fails the guard earlier, but reduces dependencies. A more detailed explanation is given in Section 4.6.

The function SmallestType returns a positive integer in bytes of the minimum type accessed in a load/store operation. The unrolling factor is heuristically determined by the smallest type and the size of the vector register. The smallest type has been chosen, to offer more opportunity to pack instructions. By choosing the biggest type, occasionally packed instructions do not fill up each slot in the vector register.

After completing unroll the method ArrayAccessInfo extracts a list of operations that reference memory (read/write) and all modifications on integer variables.

Algorithm 1 Vectorization optimization routine

```
T ... Trace loop
vs ... Size of the hardware vector register
$M_r$ ... Set of instructions that read/write memory references
$I_v$ ... Set of integer modifications
function Vectorize(T,vs)
    T ← Relax(T)
    b ← SmallestType(T)
    factor ← $\frac{vs}{b}$
    $T_u$ ← unroll(T,factor)
    $M_r$, $I_v$ ← ArrayAccessInfo($T_u$)
    G ← BuildDependecyGraph(T, $I_v$)
    $P$ ← InitPairs(G, $M_r$, $I_v$)
    $P$ ← Extend($P$, G)
    $P$ ← Combine($P$)
    $T_{vec}, savings$ ← Schedule(G, P)
    if savings $\leq -1$ then
        return T
    return $T_{vec}$
```

I_v is used to determine if memory loads/stores alias or if they are adjacent in memory e.i. Adjacent. Without inferring this information, the resulting dependency graph cannot assume that two memory stores don't depend on each other. This introduces edges which are not necessary in most cases, but prohibit vectorization. As for now building dependencies is seen as a black box. It is explained in detail in Section 4.5.

InitPairs, Extend and Schedule are shown in Algorithm 2,3,4 respectively. Isomorphic is defined as “semantically equivalent intermediate instruction”.

4.2.1 Initialize and Extend

InitPairs in Algorithm 2 creates pairs of adjacent memory operations (load/store) that are both isomorphic and independent. Only relying on the property of independence, a parallel execution is semantically valid. Isomorphism is a necessary property to ensure that SIMD instructions can execute the same operation on both instruction.

Assuming that this operation would not be a valid grouping (e.i. not semantically equivalent), then the resulting pair of operations, executed in parallel must return a different result than the sequential execution. Two operations that access adjacent memory cells would yield different results. This immediately contradicts the assumption. A similar reasoning can be applied to two store operations. Both of them are known to write to different (non overlapping) adjacent memory cells, thus the order does not alter semantics.

Algorithm 2

```
function InitPairs(G, $M_r$, $I_v$)
    $P \leftarrow \emptyset$
    for $m_1, m_2 \in M_r \times M_r$ do
        if Adjacent($m_1, m_2$) $\wedge$ Isomorphic($m_1, m_2$) $\wedge$
            Independent(G,$m_1$,$m_2$) then
            $P \leftarrow P \cup$ Pair($m_1$,$m_2$)
    return $P$
```

Extend in Algorithm 3 enumerates all known pairs and tries to follow the definition and use chains. The Cartesian product of the two calls to Def/Use represent the instructions combinations possible for one pair. The dependency graph and the edges lead to either definition of arguments or uses of the operation result. The routine continues as long as new candidate pairs are found. Extending pairs puts arguments or results at the same position in the new pair. In the next step the combination converts pairs to packs. There is no need to shuffle the elements within the packs because operands are automatically placed at the right position.

4.2.2 Combine and Schedule

Up to this point only pairs of operations have been recorded. By design pairs can overlap with other pairs. Given the two pairs (l_1,l_2) and (l_2,l_3) they can be merged into a pack of three elements (l_1, l_2, l_3). This task is accomplished by Combine. It has been omitted from the listing, since it’s implementation is straight forward. It simply compares pack by pack and merges them if the right most operation matches the left most.

The new pack (l_1, l_2, l_3) can still contain a dependency. It is not proven that l_1 and l_3 are independent. This is a rare case, but must be considered for correct scheduling.

Algorithm 3

```
function EXTEND(P, G)
    $C \leftarrow 0$
    while $C \neq |P|$ do
        $C \leftarrow |P|$
        for PAIR($i_1, i_2$)$\in P$ do
            for $i_3, i_4 \in$ USE(G,$i_1$) $\times$ USE(G,$i_1$) do
                if ISOMORPHIC($i_3, i_4$) $\wedge$ INDEPENDENT(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ PAIR($i_3$,$i_4$)
            for $i_3, i_4 \in$ DEF(G,$i_1$) $\times$ DEF(G,$i_1$) do
                if ISOMORPHIC($i_3, i_4$) $\wedge$ INDEPENDENT(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ PAIR($i_3$,$i_4$)
```

The current implementation will notice that the pack is not schedulable and emit the scalar instructions instead.

SCHEDULE in Algorihtm 4 reorders the trace instructions and emits vector operations instead of scalar operations for all packed instructions.

NEXT picks a candidate operation that is scheduleable. An operation in the dependency graph is schedulable if there are no edges that point to the operation. This is trivially true for the label operation, which starts the scheduling. If the candidate operation to be scheduled has an associated pack, all operations are transformed to a single vector operation by VECTOROPERATIONS. For this to succeed all operations of the pack must be schedulable, otherwise the current candidate is postponed. All edges to descending operations (e.i. the ones that depend on the current operation) are removed in SCHEDULED. This way the whole graph is walked reordering the instructions.

At the same time scheduling completes the reordering, UNPACKCOST tracks how often elements need to be moved individually. Section 4.7.1 covers the aspect in more detail.

4.3 Trace Unrolling

In the previous section it was stated that "basic blocks might contain enough parallel instructions" to group them together as a PAIR. In reality, these cases are rare and most of the time the program would be required to expose these parallel statements by copying the loop body.

There is no need to rewrite the program. The same result can be achieved by unrolling the loop body several times. The running example of this section is the NUMPYPY[2] kernel to calculate unary operations in Listing 4.4. It uses an iterator interface and uses a JITDRIVER to force the creation of trace loops. After passing through all optimizations eventually a trace similar to Listing 4.5 will be emitted. These operations pass through

[2]This is a subset of the NUMPY library included in PYPY

Algorithm 4

```
function SCHEDULE(PS, G)
    $S \leftarrow 0$
    $T \leftarrow \emptyset$
    $N \leftarrow$ NEXT(G,$\emptyset$)
    while $N \neq \emptyset$ do
        $O \leftarrow$ HEAD(N)
        P $\leftarrow$ PACK(PS,O)                  ▷ Retrieve the pack O is contained in
        if $\neg$ P then
            $T \leftarrow T \cup \{O\}$
            $S \leftarrow S -$ UNPACKCOST(O)
            SCHEDULED(G,O)
        else
            if PACKSCHEDULEABLE(P) then
                $S \leftarrow S -$ PACKCOST(P)
                $T \leftarrow T \cup$ VECTOROPERATIONS(P)
                $S \leftarrow S +$ ESTIMATESAVINGS(P)
                SCHEDULED(G,P)
            else
                $N \leftarrow N \cup \{O\}$      ▷ P is not yet schedulable, delay the operation
        $N \leftarrow$ NEXT(G,N)
    return T,S
```

several layers of indirection. Inlining is one of the great advantages of any tracing JIT compiler.

```
while not out_it.done(out_state):
  driver.jit_merge_point(...)
  # get the element at the iterator position
  elem = inp_it.getitem(inp_state)
  # compute the function and at the same time set the element
  out_it.setitem(out_state, func(elem))
  # advance the iterators
  out_state = out_it.next(out_state)
  inp_state = inp_it.next(inp_state)
```

Listing 4.4: NumPyPy unary kernel written in RPython. `inp_it` is the input iterator. The index position of the iterator is saved in `inp_state`.

```
label(a,b,i,n)
x = load(a, i)
y = abs(x)
store(b, i, y)
j = i + 1
guard(j < n) [a,...]
jump(a,b,j,n)
```

Listing 4.5: Equivalent optimized trace of the unary NumPyPy kernel using the `abs` operations.

```
label(a,b,i,n)
x = load(a, i)
y = abs(x)
store(b, i, y)
j = i + 1
guard(j < n) [a,...]
# rename i -> j
z = load(a, j)
w = abs(z)
store(b, j, w)
k = j + 1
guard(k < n) [a,...]
jump(a,b,k,n)
```

Listing 4.6: Unrolled trace creating a rename map at the end of each completed unroll iteration. Instructions that output a variable must create a fresh variable.

The unrolling factor is determined by the memory reference that has the smallest element size. The size of the vector register (e.i 16 bytes) divided by the smallest element size (e.g. 16-bit integer, 2 bytes) yields an unrolling factor of eight.

To get the result shown in Listing 4.6, an associative data structure is needed to remember variables that must be renamed. Every operation of the form

$$r_1, ..., r_n = op(a_1, ..., a_n)[f_1, ..., f_2]$$

is replaced by a new operation utilizing the subscript $h[key]$ and function $create(h, varold)$ using the table h. *create* allocates a fresh variable, inserts it into a hash table (using the key *varold*) and returns the new variable. $h[key]$ simply returns the table content for *key* and $h[key] = value$ assigns the *value* to the *key* of the table.

$$create(h, r_1), ..., create(h, r_n) = op(h[a_1], ..., h[a_n])[h[f_1], ..., h[f_2]]$$

The only operation that needs different treatment is the jump operation. After it's arguments have been renamed using the strategy above, the hash table h gets assigned the new values in the following way:

$$\{h[l_i] = j_i \mid \forall(j_i, l_i) \in \{(j_1, l_1), ..., (j_n, l_n)\}, l_i \in label(..., l_i, ...), j_i \in jump(..., j_i, ...)\}$$

After the loop has been unrolled once (assuming that `a` contains 64 bit integers) there are now potentially six instructions that can be executed in parallel.

4.4 Array Subscripts

AoT compilers carry the loop information in their high level IR. The loop index variable and element size are provided by the programmer and the compiler can pass this information from the AST to the optimizer. PyPys IR splits index modification among several different instructions. This can span to more than sixteen instructions for the highest unroll factor. To prove adjacency of two memory accesses it is not sufficient to know that the two indices i and j access the same array. They can either be adjacent, alias each other, overlap or are not adjacent. Missing adjacency information prohibits grouping and finally prevents the assembly of SIMD instructions. To prove adjacent memory locations the notion of memory references and variable indices are introduced.

4.4.1 Memory references and variable indices

Both terms have been briefly explained earlier. By comparing memory references and their known variable index it can be proven if memory operations are adjacent. The following example trace has the variable indices and memory references annotated in the comment.

```
label(p,a,...) # a = (a * 1/1 + 0)
x = load(p,a)  # mem. ref. (p, (a * 1/1 + 0))
b = a + 1      # b = (a * 1/1 + 1)
c = b * 4      # c = (a * 4/1 + 1)
e = load(p,c)  # mem. ref. (p, (a * 4/1 + 1))
...
```

Listing 4.7: Trace instructions that show who

A variable index is a linear equation of the from $v_1 = (i\frac{a}{b} + c)$. Two memory references m_1 and m_2 are adjacent s bytes if the access the same array and their index variables v_1 and $v_2 = (i\frac{d}{e} + f)$ satisfy the following:

1. Both use the same index variable i.
2. Coefficients divide to the same value. $\frac{a}{b} = \frac{d}{e} \wedge a \bmod b = d \bmod e = 0$
3. And the constant factor exactly matches the byte stride $|c - d| = s$.

4.5 Dependency Graph

We have seen that there are many solvers to prove dependencies. Why not reuse implementation? There are two reasons why another path has been chosen:

1. There exists no solver that is written in RPython, or would be easily integrated into the optimization pass.

2. There is no need to track dependencies across loop iterations, because the optimization will never reorder iteration instances. Only the loop independent dependencies are tracked.

The algorithm implemented for dependency construction distinguishes three cases. It tracks every definition of a new variable and remembers the defining operation.

1. **A pure or side effect free operation** only depends on the input variables. It searches for each definition of the variable and adds an edge between the definition and the use.
2. **Guarding operations** additionally depend on their fail arguments. In a similar fashion (to 1.) an edge is added from the last definition of the fail argument. Further edges prevent the following two scenarios: Operations with side effects can't be executed after the guard and guard reordering in general is not allowed.
3. **For all other operations** it must be assumed that they modify their parameters. This is equivalent as redefining the used variable at the current position.[3]

In Figure 4.2 a simple trace and the constructed data dependency graph are shown. Some transitive dependencies are omitted. The interesting cases are that instruction at line 3 depends on 4. The algorithm is not able to prove that i and j are adjacent and thus must assume they alias. Whereas 4 and 6 do not depend on each other, because k is known to be adjacent to j by one element.

4.6 Guard Relaxation

Guards specify additional arguments. Those are needed in case of a guard failure. The "blackhole" interpreter uses these arguments to fill the missing values in the stack slots and prepares the interpreter for resuming the execution.

The guard (line 5) in Listing 4.8 makes line 2 and 6 dependent. Whenever these edges are removed, the scheduler allows `store` at line 6 to be emitted before the `guard`. If the guard exits this might leave 10 at position j.

In a loop the index is most often guarded by such an instruction. This makes it impossible to vectorize a trace. A proposed solution for this problem is hereafter called "guard relaxation". It is a transformation that swaps dependency edges and force guards to exit at an earlier point in time.

Let G be an arbitrary acyclic dependency graph of a trace T. The second operation in the trace is an early exit guard[4] g_e and the trace contains a guard g_i ($g_e \neq g_i$).

Let $\pi = [s_1, ..., s_n]$ be an arbitrary path from g_i to g_e of length n. For the transformation to be semantics preserving all paths P from g_i to g_e must only contain guards or pure instructions.

[3]An exception to the rule is storing/loading to memory. An optimization tracks the redefinition of cells instead of the whole variable.

[4]This is a crafted guard instruction that allows to exit the trace without modifying any state.

```
label(p,i)
j = load(p,i)
store(p,i,3)
store(p,j,4)
k = j+1
store(p,k,5)
jump(p,i)
```

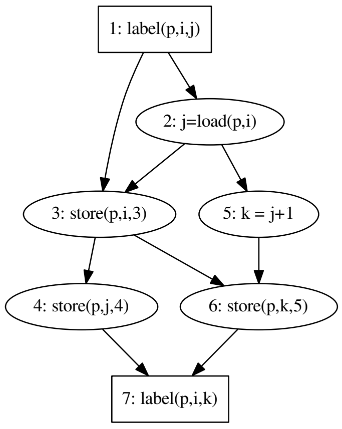


Figure 4.2: A simple trace and its data dependency graph. Some transitive dependencies are excluded.

```
label(p,i)
store(p, i, 10)
j = i + 1
b = j < 100
guard_value(b, 11) [p]
store(p, j, 10)
k = j + 1
jump(p,k)
```

Listing 4.8: A trace that contains a guard that makes the two store instructions dependent.

The following edges are inserted and removed:

1. The path segments $\pi[n-1:n]$ and $\pi[s_1:s_2]$ are removed.
2. An edge from g_i to g_e is added.
3. $\forall \pi \in P$ a new edge from the label to s_{n-1} is added.

The last assumption to make this transformation valid is that the fail arguments of g_i are exchanged with the ones of g_e. If the operation exits right after the label, no state will be modified and the BLACKHOLE interpreter is able to restore a sane context. In Section 4.5 it was mentioned that a stateful operation op_s adds a dependency edge to guard g_i if $trace_{idx}(op_s) < trace_{idx}(g_i)$ holds. If op_s does not contribute to input

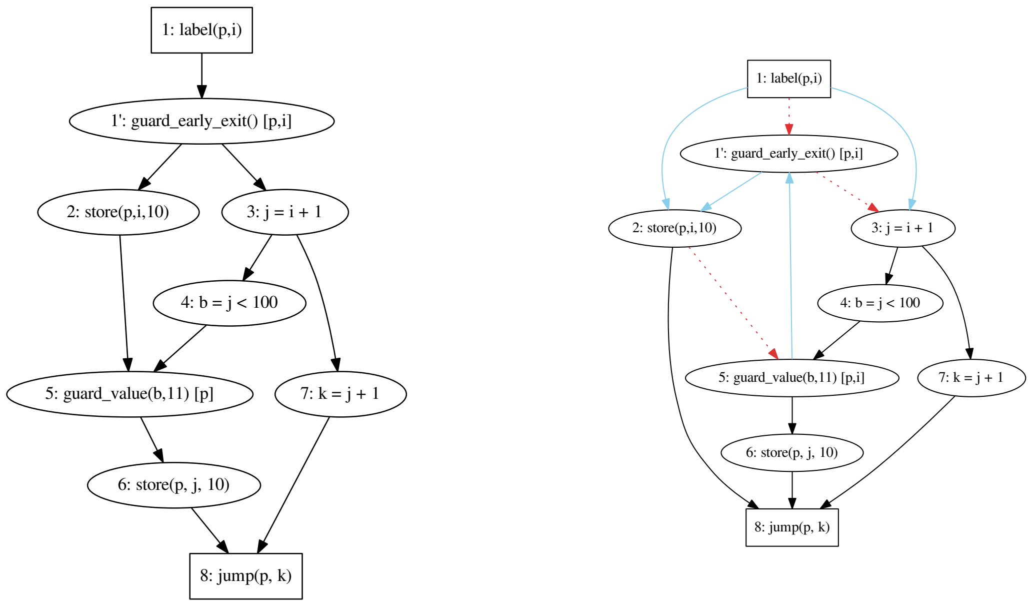


Figure 4.3: The guard relaxation transformation show for the Listing 4.8

arguments of g_i, it is possible to execute g_i at the early exit. Because no state is modified the guard fails or not, the path segment $\pi[g_i : op_s]$ can be sanely removed.

The resulting dependency graph G' must still be acyclic. Rule 1. removes edges. Removing an edge cannot make an acyclic graph cyclic. Rule 2. adds an edge from g_i to g_e. A cycle cannot be created, because the last segment $\pi[n-1:n]$ was removed and g_e is not contained in any path segment $\pi[1:n-1]$. The last rule cannot create a cycle: The label instruction is the first operation to be emitted, thus cannot have any dependency edge pointing to it.

Figure 4.3 shows the transformation before (to the left) and after (to the right) on the dependency graph. Dashed (red) edges indicate removal, light blue ones indicated that they where added to the graph. The path $\pi_0 = 3, 4, 5$ is pure. It is the one that is pulled towards the label. The edge between 2 and 5 can be removed, because the fail arguments of g_i are exchanged. This transformation hereafter is called "exit early".

4.7 Grouping, Scheduling and Assembling Instructions

Section 4.2.2 already gives some insight into this part of the algorithm. The reason for a "cost model" has not yet been given.

IR name	SIMD ops.	Savings
load float32	`MOVUPS`	3
unpack high float64	`UNPCKHPD`, `SHUFLPD`	-2
unpack low float64	`MOVSD` or `UNPCKLPD`	-1
add int16	`PADDSB`	7

Table 4.1: This shows the savings for some example operation of the intermediate representation. Note that the unpacking of the low 64-bit floating point does not need any instruction in the assembler backend. The SSA form of the trace creates a new variable after modification and forces the register allocator to copy the vector register. Nonetheless its savings are negative to count down the threshold counter.

4.7.1 Cost model

Classical vectorization is able to detect loop carried backwards dependencies. Those prevent the generation of vector statements. There are no loop carried dependencies known to VecOpt. The cost model in the current implementation helps to identify non transformable sequential loops. Unpacking operations indicate whether the packing of instructions gains speed or can slow down the execution.

If a loop statement carries dependencies e.g. `a[i] = a[i-1] + a[i]` this leads to inefficient unpacking/packing instructions. Note that this loop requires one iteration to finish before the next can start. Most of the loops that should not be transformed can by addressed by preventing one simple pattern: Only memory load operations can be packed. The slightly more elaborate approach added to VecOpt models SIMD instructions in terms of costs and benefit.

The scheduling algorithm is interwoven with logic to estimate the "savings" of the loop. The formula $savings = -cost + count(pack) * \text{benefit}$ is the basis for each instruction.

Table 4.1 shows the savings for different operations. Note that depending on the vector slot position the penalty might be more significant for different CPUs. E.g. Unpacking the higher element of a double precision floating point has a higher cost than unpacking the lower element[5].

During scheduling a variable keeps track of the overall savings. The vectorizer bails if the savings at the end go below the threshold (default 0).

4.7.2 Combination and Scheduling

Combination maximizes the packs and removes all related pairs. Heuristically the packs are split at the position that fills the last slot of the vector register. This might leave some instructions in the pack which are scheduled in the scalar version of the operation.

Scheduling adds a priority to some nodes. During the dependency construction it is known that some instructions must be emitted before a full pack can be scheduled.

[5]The x86 assembler backend needs at least two assembler instructions for a double precision high slot. The lower slot need at most one.

E.g. all nodes that have been relaxed to an earlier position. These are then marked with a higher priority and the scheduler will emit them as soon as their dependencies are solved. This prevents the scheduler to check on pack items until all their dependencies are emitted.

4.7.3 Assembling

The final assembly stage has been adapted to emit SSE4.1 instructions for the vector IR instructions. PyPys register allocator uses the linear scan heuristic initially proposed by Poletto and Sarkar [PS99] in 1999. The sorted live ranges with ascending start time are assigned a new register each at a time. Whenever no register is available the heuristic spills the active live range that ends at the latest point in the trace.

Code generation only maps one intermediate instruction to zero or many CPU instructions. It is tightly integrated with the allocator. It well deserves the name assembler, because it only assembles the trace. Techniques such as tree pattern matching or other optimization techniques to aid the code generation are not applied.

4.8 Complexity

The big O notation is used to express the complexity. Table 4.2 summarizes the memory and time complexity of each sub part. A dash in the third column indicates that no memory is allocated, but only freed.

Complexity is most of the time expressed by the number of instructions. n denotes the total number in a trace. It is hard to give a rough estimate of arguments[6] for each individual instruction. Quantifying them over a whole trace is even harder. Thus the maximum number is assumed in a range from eight to fifteen. Since they are bounded and usually very small they are assumed a constant factor per instruction.

The **dependency construction** passes two times over the list of trace. The basic data structure is a hash map where it keeps track of definitions. The following lists the operations needed to construct the graph (split over three passes).

- Definition, redefinition and usage of a new variable. Complexity of lookup and insertion is logarithmic in the size of the hash map.
- Guard. It maintains the guard order by adding one edge to the previously known guard and edges to non pure operations that precede the guard. At the worst case all previous operation could be non pure, resulting the complexity $O(n)$. Nevertheless it is assumed constant. For each guarding instruction, there are usually at most two non pure operations.

This would give an approximation of $O(n \times \log v_{max})$. v_{max} denotes the maximum number of variables contained in the hash map. It is clearly dominated by the number of instructions. This yields a complexity of $O(n)$.

[6]In this context arguments include the result and fail arguments.

Sub problem	Complexity	Memory Complexity
Dependency construction	$O(n)$	$O(n+e)$
Unrolling	$O(n*u)$	$O(n*u)$
Adjacent memory	$O(m^2)$	$O(p)$
Packset extension	$O(p^2)$	$O(p_e)$
Combination	$O((p+p_e)^2)$	-
Scheduling	$O(n)$	$O(v)$

Table 4.2: Complexity in big O notation for several sub parts of the algorithm.

The graph needs at least n new node objects and at most e edges. The connection density other than the ones of the label is considered sparse, thus an adjacent list representation has been chosen to save space. At the very least $\Omega(n)$ node objects and $\Omega(n-1)$ edge objects are allocated. In the worst case the number of edges depends on the definition use chains and the state modifying operations. It has been observed that the number of dependencies is a constant factor for each node (between one to four edges). Thus it is assumed to be linearly correlated to the number of nodes, written as $O(e)$.

Unrolling iterates the trace loop by an unroll factor u. Additional steps are needed to copy the arguments and correctly renamed them. All operations are copied, thus $O(n*u)$ new operation objects are allocated.

At first recorded **memory references** are compared to each other memory reference. If they are proven to be adjacent, a new pair is allocated for them. Clearly this is quadratic in the number of memory references $O(m^2)$. At most $p = \frac{m}{2}$ new PAIRS are allocated. **Packset extension** follows both definitions and used variables of a previously created PAIR. The possibilities to follow are very limited. As mentioned earlier, the graph is rather sparse. The amount of pack comparisons dominate $O(p^2)$. p_e denotes the number of newly created PAIRS. **Combination** subsequently merges pairs into packs. This phase takes $O((p+p_e)^2)$ steps to compare each combination. **Scheduling** starts at the label node and adds each graph node to the final list. This removes the edges from the graph. In the worst case the scheduler walks every node in the graph once and appends the operation to the new list. The `Next` function needs to hold back instructions that are contained in a pack. Thus in the worst case it would require to delay $p_c - 1$ nodes (p_c denotes the number of instructions in the pack) before it can again try to schedule the pack. Since pack size is bounded by 16, it is assumed constant.

Notwithstanding the fact that there is quadratic complexity in the worst case runtime for some parts of the algorithm, usually these factor play a minor role in the overall runtime. It is claimed that under normal circumstances the lower bound complexity is linear and does not deviate much.

Assumptions are needed to claim the time complexity shown in Table 4.2. To show that in practice these assumptions are indeed valid, Section 6 in Table 6.1 shows the microseconds needed by the optimization.

4.9 Limitations

One of the most limiting boundaries is the **trace**. It is “only” a linear sequence of instructions. A region based or even control flow based input could yield better opportunity for the algorithm. It is however impossible to do in a tracing JIT compiler.

The **stride** that advances the array index **must be constant**. In any other case it must be assumed that memory load/store operation alias to the same memory cell. RPython provides a convenient annotation to promote a variable to a constant. Using this utility the trace automatically contains the stride as a constant.

Whenever memory is copied frequently in NumPy expressions, the reference counting technique of CPython is able to free memory much more quickly than the advanced generational mark and sweep garbage collector of PyPy. A very simple solution would refactor the internal structure of the NumPyPy source code to faster free big chunks of memory which are created very frequently by the NumPy implementation.

The isomorphic definition is restricting the pairs creation. Considering a case where $j = i + 1, k = i - 1$ these instructions could still be isomorphic (in the sense of equivalent operation semantics) if the sign of the constant factor is inverted at one instruction and the right operator is adapted at the same instruction.

The combination stage splits the packs at the index that fills the vector register. When pairs are created, there could be opportunity to add just a single shuffle operation to allow the discovery of pairs that could not be used before that. SIMD vector extensions allow efficient shuffling of the operations and this could further widen the chances to successfully optimize a loop kernel.

Section 7.1 contains some additional limitations and more suggestions to improve the implementation even further.

CHAPTER 5

Advanced Extensions

This chapter describes additional extensions and modifications to the PyPy project to enhance the generated vector trace.

5.1 Constant & Variable Expansion

Expanding either constants or variables is a technique to reuse values located in vector registers. Given the statement $v_2 = v_1 + \mathtt{c}$ [1] SSE4.1 requires both operands for addition to be in a vector register. Constant/Variable expansion scatters the value of c to each slot of the vector register v_c. v_c is called the expanded vector register of c.

A naive implementation in SSE4.1 would expand the value at each loop iteration. This way, the performance gain will not be significant. In the worst case c will be located on the stack or in heap memory. This will require an additional memory load every iteration.

Instead, the expansion in PyPy adds a dedicated register and moves the construction before the loop is entered. This way the expansion of c is loop invariant and blocks one vector register for the whole loop.

Figure 5.1 shows an example. c is expanded in to vector register v_c and added as an argument to the label and the jump operation. In any case this increases the register pressure for vector registers.

However it is not possible to expand variables that are generated in the loop body. In this case the implementation will still expand the variable just before it is used.

5.2 Accumulator Splitting

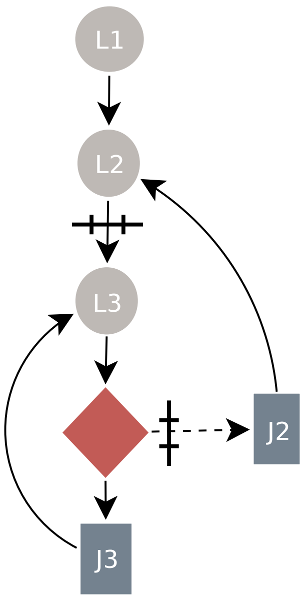
Up to this point only vector operations have been considered. Reduction patterns are also very common operations. Examples are sum, product, all and any.

[1] v_1, v_2 are both vector registers, c is a constant/variable

```
label(a,i,c)
...
x = load(a,i)
y = load(a,i+1)
e = x + c
d = y + c
store(a,i,e)
store(a,i+1,d)
...
```

```
v_c = vec_expand(c,2)
label(a,i,v_c)
...
v_1 = vec_load(a,i,2)
v_2 = v_1 + v_c
vec_store(a,i,v_2,2)
...
```

Figure 5.1: Constant expansion on a trace sequence. Left hand side shows an unrolled trace. On the right the constant expanded trace is shown.



```
    label(p,i,a) # 1
    # fresh vector register
    # all bits zeroed
    v_0 = vec_empty(2)
    # set v_0[0] = a
    v_a = vec_pack(v_0,a,0)
    label(p,i,v_a) # 2
    guard(i < n) [...]
    v_1 = load(p,i,2)
    v_2 = v_a + v_1
    jump(p,i+2,v_2) # to 2
```

Figure 5.2: Points which need special modification for accumulating values. See Figure 3.3 for the shape description. The control flow from L1 to L2 forms the first iteration of the loop, L2 to J3 the optimized vector loop and the guard to J2 a bridge.

The fundamental property that allows this transformation is called "commutative". A binary function f is called commutative $f(x, y) = f(y, x) \quad \forall(x, z) \in (D \times D)$ in the domain D. Addition $(+)$, Multiplication $(*)$ and Logical and $(\wedge)$, or $(\vee)$ or xor $(\oplus)$ are examples of commutative binary functions.

Similar to the expansion in Section 5.1, reduction patterns need additional initialization of a dedicated reduction register v_r and a finalization step to handle the last out standing reduction operation.

Figure 5.2 indicates the points in the trace tree that need special adjustment to correctly accumulate values. To share the modifications for the entry point, the original label (L2) is preserved. A new label (L3) separates the first few instructions to setup the vector register and is the target of the backward jump J3. Any bridge that is generated in the loop and is able to reenter the optimized trace loop starting from label L3, must enter L2 instead.

The algorithm described in the second part of Section 4.1 needs adaption. Reduction carries dependencies and would prevent the creation of a pair.

Listing 5.1 shows an accumulation sequence. The pattern matches only the following sequence:

1. An accumulation variable a_0 provided by the label is used at position i of a commutative operation op_1. Position[2] j uses a variable l_1 loaded from a memory location. This condition triggers the pattern recognition.

2. The output variable a_n of op_n (a_1 and op_1 in this example) is used at the same position i in another commutative operation op_{n+1}. Position j uses another variable l_{n+1} loaded from memory. Both op_n and op_{n+1} form a new accumulation pair if and only if l_n and l_{n+1} are loaded from adjacent memory locations and share the isomorphic property every pair must adhere to.

3. If more pair candidates can be found matching op_{n+1}, step 2. is repeated until no further accumulation pairs can be found.

4. Every operation op_n and their output variable a_n is either used in a commutative operation or is an argument of the final jump operation at the same position a_0 originated from.

The scheduling algorithm continues, but ignores the dependency between the elements of the accumulator operations.

If the resulting value of the accumulation pair is required the accumulator needs to be “flushed”. In this example the value needs to be added horizontally and is stored in the final destination register whenever the guarding operation exists. In the current implementation an accumulation flush is only allowed to happen at guard exit. Any other case will remove the accumulation pair/pack from the packset and proceed.

5.2.1 Any/All Reduction

Using the reduction optimization above `any` and `all` can be easily implemented with the logical operators $\wedge$ and $\vee$ in a vectorized form. Yet, it is not always necessary to scan the full array to come to the conclusion that `all` will return `False`. At any vector element that equals zero (for numerical arrays) the loop can be exited. Thus instead of accumulating the value a guard instruction tests if the value is true. PyPy's IR has been enhanced to guard vector registers. Analogous to `all`, `any` can escape the trace when it encounters a non zero value.

[2]Indeed there are only two positions i, j for a binary function.

```
1 label(..., a_0)
2 ...
3 a_1 = l_1 + a_0
4 ...
5 a_n = l_n + a_1
6 ...
7 jump(..., a_n)
```

Listing 5.1: The sequence after unrolling summation once. l_x describes the value loaded from memory, a_x is the accumulation variable. Instruction at line three and five are dependent (δ).

5.3 Trace Versioning

Entering a trace produced by VecOpt already passed one iteration of the loop. If the iteration length and index stride are both even, the last iteration switches back to the interpreter. This can happen roughly hundred times before a bridge is compiled out of the failing guard.

To prevent the switch to the interpreter before the loop has ended, a snapshot of the original scalar trace can be taken. This version can then later be associated with a guarding instruction. A loop version can be compiled and stitched to the vector trace before the trace is entered the first time. To preserve semantics, not all guarding instructions are allowed to version the trace, but only guards that are exited early.

Trace versioning has another neat use case for this optimization. If the target JIT architecture does only allow unaligned memory access, a special guard can ensure that accesses to memory locations is aligned through out the whole trace. At present only unaligned memory access is emitted. Vector extensions normally support unaligned memory access. The performance penalty on modern platforms is minor compared to earlier versions. Section 7.1 contains other solutions how to use aligned memory access, but all of them need trace versioning to ensure that the operating system does not terminate the program on unaligned memory access.

5.4 Guard Strength Reduction

Guarding instructions are omnipresent in traces. Not only do they protect the looping condition or conditions in the program flow, but they ensure that type information does not change. SPUR [BBF^{+}10] for example implements a technique called guard “strengthening” and “implication”.

PyPy implements strengthening and implication, but does not consider arithmetic expressions protected by a guard. VecOpt duplicates guarding instruction with every

Guard implication	Satisfying equation
$guard(x + b < y + c) \rightarrow guard(x + d < y + e)$	$b \geq d \wedge c \leq e$
$guard(x + b > y + c) \rightarrow guard(x + d > y + e)$	$b \leq d \wedge c \geq e$

Table 5.1: x, y are variables, b, c, d, e are constants. Shows the equations that must hold to assume that the implication is a tautology. The multiplicative factor in the index variables has been omitted.

unrolling step. Looping overhead is not removed, but it is necessary. The index value is checked several times, incrementing it for each check. The strongest guard is the one that advances the index the most and ensures that the index does not overflow. To mitigate these effects a new optimization pass has been added after vectoring a trace. It strengthens and implies "arithmetic" guards.

The linear combinations for index variables is gathered not only for memory accesses, but for any integer variable modification. I_v denotes the index variables in Algorithm 1. Passing this information can remove guarding instructions. Table 5.1 shows two implications and the equations that must hold. Given two guards at positions i and j $(i < j)$. Whether the guard g_i implies g_j, g_j can be omitted. If $g_j \rightarrow g_i$ then g_j can be executed at the position i and g_i is removed.

5.5 Array Bound Check Optimization

Not only NumPy arrays but also instances of the array class in the Python standard library store homogeneous primitive data. Even PyPy's lists store homogeneous primitive data in the same way the array class does. This optimization is called "storage strategy" described in [BDT13].

Basic block vectorization is powerful enough to transform any trace, not just the loops in the NumPyPy library. Array instances can be modified or read using array indexing. In case of an access that is not within the array an `IndexError` is raised. Well written programs do not rely on these exceptions, but ensure that the index does not overflow the array bounds.

Array Bound Check (ABC) removal is a well studied problem and has several proposed solutions. [BGS00] build a separate graph structure using the SSA representation of the program. Several constraints are tracked such as array length, variable increments and array bound checks. A constraint solver tries to prove that some array bound checks are partially or totally redundant.

Asuru [Asu92] defines range check information at the beginning and end of a basic block. These are considered generated range checks by the basic block. Several tests prove a bound check totally redundant. A local elimination step removes redundant checks within the basic block.

There are many other solutions that are able to remove array bound checks. The ABC removal added to PyPy is a code motion technique strongly related to the guard strengthening.

```
...
guard(i < n)
...
guard(i < len(p))
a = load(p,i)
...
```

Listing 5.2: An ABC for loading from a memory location `p`.

Listing 5.2 shows a trace that contains the index guard and a length check just before the load operation. Well written loops will never trigger the second guard. Thus it must hold that $n \leq l_p$ ($l_p = len(p)$). The essence of this ABC removal adds a new guarding instruction `guard(n < len(p))` before the loop body. The following equations make this more explicit:

$$\begin{aligned} n \leq l_p \wedge i < n \rightarrow i < l_p \\ = i < n \leq l_p \rightarrow i < l_p \end{aligned}$$

Knowing that the array is at least as big as the maximum loop bound, the array bound check is redundant if it passed once before the trace loop is entered and both the l_p and n are not modified in the loop body. To ensure that the guard failure is handled without endeavoring the blackhole interpreter the loop version before the ABC optimization is stitched to the artificial guards.

CHAPTER 6

Evaluation

The evaluation is split into two parts. The first measures the time spent in the trace loops on a micro level and compares it to the scalar version of the trace. The second shows achievements of the auto vectorizer in a real world setting.

6.1 Environment Setup

The following table shows the configuration the benchmark where evaluated on.

Name	Operating System	CPU	GHz	Cores	RAM
mobile	Linux 4.0.6, 64-bit	Intel i7-4550U	1.5	4	8 GB
desktop	Linux 4.1.5, 64-bit	Intel i7-2600	3.4	4	8 GB

Although different versions of the operating system where used, it is believed that this does not influence the outcome of the evaluation.

The source code can be found in the branch “vecopt” and is based on the PyPy release version 2.6.0. The repository is located at `https://bitbucket.org/pypy/pypy`.

Section 6.2 and 6.3 use the configuration “mobile”. The revision of the source code is **f0a1bf333b2f**. For all other benchmarks “desktop” has been used.

6.2 Optimization Time

For the following measurements, the tracer and JIT compiler has been instrumented[1] to measure the time elapsed in traces. The function *clock_gettime* used in the evaluation records the CPU time spent in the process.

[1]The revision *a026d96015e4* was used for this benchmark run. It imposes a significant performance penalty when exiting or entering traces.

Count	Instruction count	Unroll factor	Microseconds	Variance
6	12-16	2	101.47	9.90
5	17-19	4	158.46	4.57
2	17	8	224.03	2.20
2	17	16	396.60	1.24

Table 6.1: Optimization time measured. Instruction count is the number before the transformation and unrolling has been applied.

The standard garbage collector "incminimark" was prevented to be run in the trace loop benchmarks by setting the minimum memory threshold to 4GB of allocated memory. Below the modified threshold, the garbage collector does not start a collection run.

Table 6.1 shows the micro seconds that have been spent in the optimization pass. It excludes all other optimizations.

6.3 Trace Loop Speedup

Figure 6.1 shows several different vector calculations. The horizontal line shows the baseline of the normal trace. Every program run iterates the operation for 1000 iterations. The vectors are sized four times the tracing threshold. The following listing shows a sample program that is used in Figure 6.1.

```
def bench(vector_a, vector_b):
  for i in range(1000):
    numpy.multiply(vector_a, vector_b, out=vector_a)
```

Single floating point operations don't show a significant speedup to their scalar trace loops. PyPy always converts floating point operations to the biggest floating point type available. The language semantics of Python make the size of floating point numbers platform specific, thus the tracer does not emit floating point operations for single floats, but casts them to double floats.

The maximum speedup can be observed for double floats multiply. Whereas others show up to half of the expected execution time. Considering that it is currently not possible to use aligned vector statements the results are quite satisfying.

Integer addition for 16/8 bits don't show very good results. It has been observed that on bigger vector sizes these data types perform better. In any case these instructions are not expected to be used very frequently in NumPy programs.

6.4 NumPy Benchmark Suite

The following evaluates VecOpt on small to medium sized numerical kernels. For these benchmarks a hardware configuration was used with a higher clock cycle than "mobile" configuration. Python 2.7.10 and NumPy version v1.9.2rc1 has been used as a base line

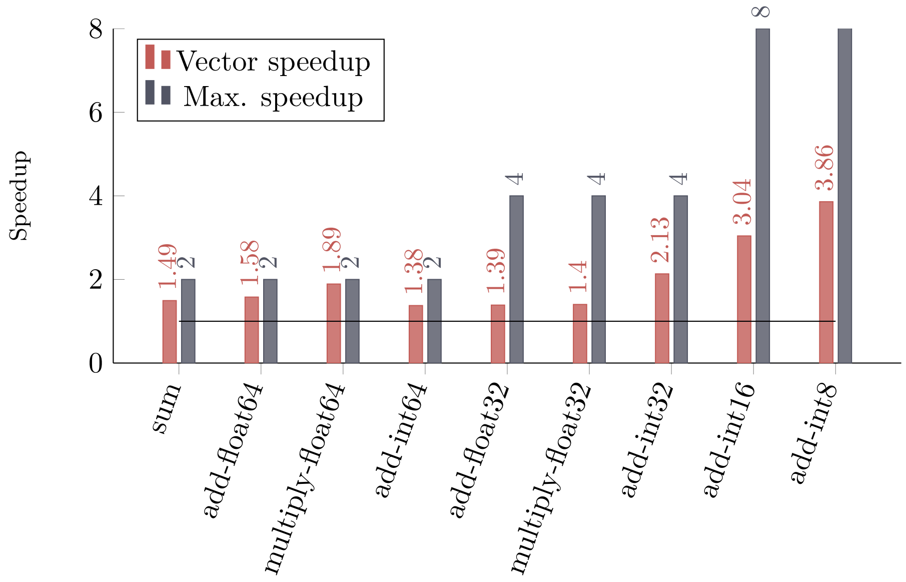


Figure 6.1: Speedup of the vectorized trace loops. Horizontal line is the baseline for the calculated speedup values ($speedup = \frac{scalar}{vector}$).

Name	Loop	Warm up
diffusion	20	5
allpairs-distances	30	20
vibr-energy	100	20
l2norm	100	20
rosen	30	10

Table 6.2: The loop count and warm up iteration count for the benchmark programs in 6.3. All kernels that are not listed loop 50 times and warm up 20 iterations.

implementation. VecOpt uses revision **3742fae37** and the forked NumPy branch for PyPy (**504ee4757**). PyPy uses the 2.6.0 release binary (**295ee98b69**).

Table 6.3 shows a NumPy benchmark suite[2]. The source code was forked and modified. The modifications executes the kernel several times to warm up the JIT compiler. Each benchmark is repeated five times and the mean value is displayed in the table. For PyPy the benchmark kernel is executed twenty times in the warm up phase. Table 6.2 shows the loop count of the kernels.

Table 6.3 shows that for some benchmarks minor improvements can be achieved. The current weakness both PyPy and VecOpt suffers from is related to allocating memory in the benchmark kernel. CPython's GC uses reference counting which immediately frees NumPy arrays. PyPy's GC might keep memory for many more cycles. In Section 6.5

[2] `https://github.com/planrich/numpy-benchmarks`. Aug. 2015

Name	CPython (C_1)	PyPy (C_2)	VecOpt (C_3)	$Speedup\frac{C_1}{C_3}$	$Speedup\frac{C_2}{C_3}$
allpairs-distances	0.9868	2.57	2.534	0.39	1.0
allpairs-distances-loops	1.826	4.287	4.177	0.44	1.0
arc-distance	0.07898	0.1813	0.1608	0.49	**1.1**
diffusion	0.5603	5.665	3.889	0.14	**1.5**
evolve	0.1967	1.815	1.728	0.11	**1.1**
fft	0.9507	0.2981	0.2955	3.2	1.0
harris	0.3485	3.119	1.504	0.23	**2.1**
l2norm	0.564	1.73	1.634	0.35	**1.1**
lstsqr	0.3844	1.506	1.39	0.28	**1.1**
multiple-sum	0.1432	0.6341	0.5768	0.25	**1.1**
rosen	0.5795	3.498	3.438	0.17	1.0
specialconvolve	0.4713	3.876	2.649	0.18	**1.5**
vibr-energy	0.2784	0.7552	0.699	0.4	**1.1**
wave	2.191	1.114	1.166	1.9	0.96
wdist	2.927	1.202	1.179	2.5	1.0

Table 6.3: Benchmark suite. C_1, C_2 and C_3 show the CPU clock time spent. C_4 and C_5 show the speedup. C_5 additionally marks the improvements introduced by VecOpt.

other kernels show the speedup when no memory is allocated within the kernel loop.

Table 6.3 indicates in column $Speedup\frac{C1}{C3}$ that CPython most of the time is a better choice than PyPy. The only reason why CPython executes faster is because a significant fraction of time is spent in native code, removing all interpretative overhead.

Furthermore note that the NumPyPy library has not completed to implement all features offered by NumPy. After fully completing the NumPyPy implementation and solving the GC issues, it is expected to reach the native performance NumPy currently offers.

6.5 Real World Program Evaluation

To show that there are really more significant improvements than presented in the previous section, a list of benchmarks has been compiled[3]:

- **som** - Self Organizing Maps[4].
- **dot** - Matrix vector dot product.

[3]`https://github.com/planrich/pypy-simd-benchmark` Aug. 2015

[4]A numeric application that makes heavy use of vector subtractions, multiplications, distance and summation. Similar to principal component analysis this procedure can be employed as a pre step for machine learning. This implementation is not complete. It only simulates the "find nearest neighbor" and "update weight vector" step of the algorithm.

Name	Vector size	Repeat count
som	256	4.000
dot	1.000	1.000
any	1.024	1.000
add*	2.500	10.000
sum*	2.500	10.000
fir*	200	3.000
rgbtoyuv*	$1024 * 768$	500

Table 6.4: The vector size and the repetition count of the kernel benchmark programs in Figure 6.2. All programs are run ten times and the mean value is used to calculate the speedup value.

- **any** - Micro benchmark stressing the any NumPy operation.
- **fir*** - Finite impulse response.
- **add*** - Addition of a Python array.
- **sum*** - Summation of a Python array.
- **rgbtoyuv*** - RGB to Y'UV conversion using Python arrays.

All benchmarks that end with an asterisk symbol (*) are pure Python implementations. Indeed the optimizer makes no distinction between NumPy and Python traces, but is currently by default deactivated for the latter.

The benchmark programs shown in Figure 6.2 are included in the Appendix 7.2. Opposed to the previous section no memory is allocated in the loop body. Indeed the first three benchmarks show that the GC issues do not occur when no memory is allocated in the loop. This benchmark as well proves that the optimization is applicable even for user traces. There are some limitations that are planned to be ironed out in the next major release of PyPy. Still they show significant speedup. As noted in the Figure description, the runtime of Python has been excluded for the last four benchmarks they take far too long to complete.

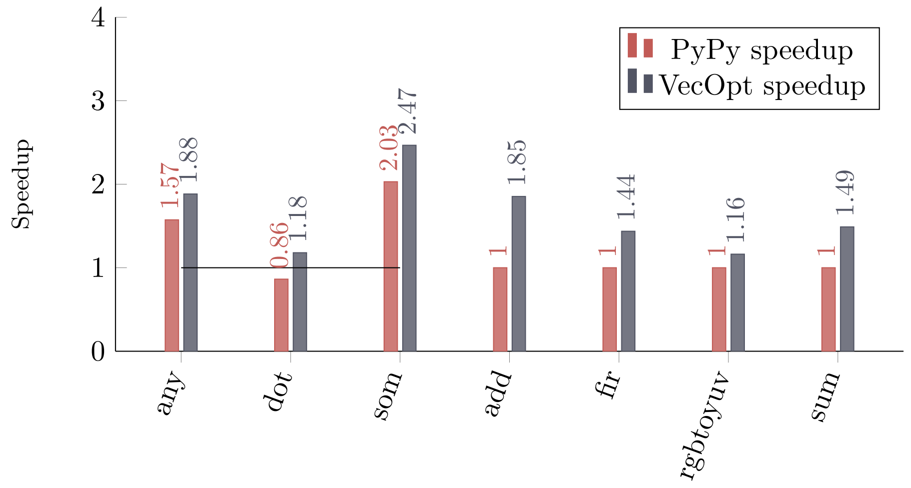


Figure 6.2: Yet another benchmark plotting the speedup. The first four runs use CPython as base line to measure the speedup (Indicated by the horizontal line). For all others CPython had to be excluded from the benchmark run. All of them are written in pure Python. CPython is not able to execute any computation in native code and thus takes far to long to complete the benchmark run. The speedup of VecOpt in these cases uses PyPy as baseline implementation. Due to some limitations of the current prototype, RGB to YUV operations on floating points rather than 8/16 bit bytes.

CHAPTER 7

Conclusion

7.1 Future work

In this section a look beyond the bounds of this thesis is taken.

PyPy is under going constant changes and improvements. Only recently work has been started to cut down the optimization time and improve the warm up speed of the virtual machine. Interestingly these changes already track definition use of the IR operations more directly and with little effort, the dependency construction step can be merged with the new model. There are plans to integrate these changes and enable them by default with the new optimization setup.

PyPy's next major release will merge both the new optimization model with the auto vectorizer into the production release. With this enhanced model Python loops that access lists can be additional source for this optimization.

One deficiency has already been mentioned in the evaluation section. Allocating memory frequently is not handled very well by PyPy's GC. Moreover it does not know that the allocated NumPyPy array is a large chunk of memory, but only sees the object encapsulating the pointer to the actual storage. This problem could be mitigated by changing parts of the NumPyPy library.

The unrolling heuristic performs very well for most numerical loops. But it is ignoring the fact that there could be already several memory load/store instructions that could be grouped to one vector operation. By enhancing this heuristics, the programmer could specify the parallel instructions manually, preventing additional loop unrolling.

To date there are the bare essentials to version different traces, assemble and stitch the trace tree. Aligned memory operations are both necessary for some platforms and more efficient on others. There are currently two feasible solutions to this limitation. First, an additional unrolling step could try to realign the load/store operations. Till now, at least one iteration of the loop body is always executed before entering the vector loop. Artificial guards would ensure that memory operations are aligned. If the criteria is not met control flow would bail into the unaligned vector loop.

Second, by enhancing the heuristic to determine the initial unrolling of the scalar body, the programmer could manually specify the body several times. At the position just before the vector loop, chances are high that memory can be loaded/stored aligned. Similar to the first suggestion guards ensure the alignment.

Some loop bodies require shuffling of the elements in the vector register. A common example would be interleaved formats (e.g. RGB). A single shuffle could extract all red values from the vector register. Currently the scheduler would unpack and pack the elements into a new register one by one, eventually canceling the vectorization process.

7.1.1 GPU Programming model

GPU programming model is different to SIMD instructions. GPU's are co processors specifically tailored for rendering millions of triangles in a fraction of a millisecond. In the last decade a new term "General Purpose GPU" (GPGPU) has formed with standards such as OpenCL [SGS10] or other proprietary solutions to be able to use the massive parallel design of graphics cards for non graphic related problems.

The VecOpt's design is tailored to short SIMD instructions. Nevertheless it is believed that it is possible to enhance RPython to make use of these powerful devices. The following gives a vision and some coarse grained steps on how to implement a first prototype.

RPython is a subset of the Python language. By further restricting RPython would let the VM designer write modules that can be executed automatically on a GPU device. Instead of transforming the annotated and rtyped CFG to C, it could be translated to intermediate format or OpenCL kernels.

7.2 Summary

PyPy's new auto vectorizer provides evidence that even a dynamically typed language such as Python provides enough runtime information to optimize numerical kernels.
A preprocessing step moves guards to an earlier position in the trace and unrolls the loop body to ensure that enough parallel instructions are available. The algorithm creates groups of parallel and isomorphic instructions and uses this meta information to reschedule the trace. Groups emit special vector instructions forcing the assembler to write SSE4.1 instructions. PyPy's list strategies store primitive data without embedding it into an object. By adopting the new optimizer model it will be possible to use SIMD instructions for loops that read and modify from Python lists. This is currently restricted to the array module in the Python standard library.
VecOpt transformation time is fast and the implementation complexity is modest. RPython is a versatile tool chain to write byte code interpreters and the new auto vectorizer is available to many virtual machines and yet more to come.

Code Samples

```python
def bmu(grid, x, y, sel, tmp):
    """ Entry in the grid that has the smallest euclidean
   distance. """
    min_pos = 0
    min_dist = float(10000)
    for i in range(x*y):
        entry = grid[i]
        ufunc.subtract(sel, entry, out=tmp)
        # euclidean dist
        ufunc.multiply(tmp, tmp, out=tmp)
        d = ufunc.sqrt(tmp.sum())
        if d < min_dist:
            min_dist = d
            min_pos = i
    return min_pos

def adjust(G, X, Y, pos, hci, alpha, selection, tmp):
    vector = G[pos]
    # vector = vector + alpha * (vector - sel)
    ufunc.subtract(vector, selection, out=tmp)
    ufunc.multiply(tmp, alpha, out=tmp)
    ufunc.add(vector, tmp, out=vector)
    G[pos] = vector

def som(D,I,X,Y,G,DATA,tmp):
    r = random.Random()
    r.seed(0)
    for j in range(I):
        i = r.randint(0,49)
        selection = DATA[i]
        pos = bmu(G, X, Y, selection, tmp)
        alpha = float(j)/I
        hci = 1 # not calculated for this bench
        adjust(G,X,Y,pos,hci,alpha,selection,tmp)
    return None
```

Listing 1: Self Organizing Maps.

```python
def conv_rgb_to_yuv(R,G,B,Y,U,V):
    # see https://en.wikipedia.org/wiki/YUV
    # a naive implementation
    i = 0
    while i < len(R):
        Y[i] = R[i] * 0.299 + G[i] * 0.587 + B[i] * 0.114
        U[i] = -0.147 * R[i] + -0.289 + G[i] + 0.436 * B[i]
        V[i] = 0.615 * R[i] * -0.515 + G[i] * 0.100 + B[i]
        i += 1
```

Listing 2: RGB to Y'UV conversion.

```python
def fir(B,X,n):
    # see https://en.wikipedia.org/wiki/Finite_impulse_response
    i = 0
    res = 0
    while i < n:
        b = B[i]
        x = X[i]
        a = x*b
        res += a
        del b,x,a
        i += 1
    return res
```

Listing 3: Finite impulse response.

```python
def numpy_dot(M,V,O,count):
    for _ in range(count):
        np.dot(M,V,out=O)
```

Listing 4: Matrix vector dot product.

```python
def py_add(A,B,size):
    i = 0
    while i < size:
        A[i] = A[i] + B[i]
        i += 1
```

Listing 5: A pure Python implementation for vector addition.

```python
def py_sum(V,size):
    i = 0
    a = 0
    while i < size:
        a += V[i]
        i += 1
    return a
```

Listing 6: A pure Python implementation for summing up a vector.

```python
def numpy_any(V,count):
    # V contains only 0s
    for _ in range(count):
        a = V.any()
        assert not a
```

Listing 7: Any NumPy operation.

Bibliography

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64. ACM, 2007.

[ABF12] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. Loop-aware optimizations in PyPy's tracing JIT. In *ACM SIGPLAN Notices*, volume 48, pages 63–72. ACM, 2012.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:491–542, 1987.

[ASRV07] Mauricio Alvarez, Esther Salamí, Alex Ramírez, and Mateo Valero. Performance impact of unaligned memory operations in SIMD extensions for video codec applications. In *ISPASS'07: IEEE International Symmposium on Performance Analysis of Systems and Software*, pages 62–71. IEEE Computer Society, 2007.

[Asu92] Jonathan M Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):109–118, 1992.

[Ban88] Utpal Banerjee. Dependence tests. In *Dependence Analysis for Supercomputing*, pages 101–148. Springer, 1988.

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[BBF$^+$10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *ACM Sigplan Notices*, volume 45, pages 708–725. ACM, 2010.

[BCW$^+$10] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 59–68. ACM, 2010.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[BDT13] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *ACM SIGPLAN Notices*, volume 48, pages 167–182. ACM, 2013.

[BDT15] Edd Barrett, Lukas Diekmann, and Laurence Tratt. Fine-grained Language Composition. *arXiv preprint arXiv:1503.08623*, 2015.

[BGGT02] Aart JC Bik, Milind Girkar, Paul M Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel® architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[BLS10] Carl Friedrich Bolz, Michael Leuschel, and David Schneider. Towards a jitting vm for prolog execution. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 99–108. ACM, 2010.

[Bol13] Carl Friedrich Bolz. *Meta-tracing Just-in-time Compilation for RPython.* PhD thesis, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2013.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[CL97] Gerald Cheong and Monica Lam. An optimizer for multimedia instruction sets. *Contract*, 30602(95-C):0098, 1997.

[Com] PyPy Community. PyPy project website. `http://pypy.org` Sep. 2015.

[CSR+09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2009.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Notices*, volume 24, pages 146–160. ACM, 1989.

[Cun10] Antonio Cuni. *High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages.* PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, 2010. Technical Report DISI-TH-2010-05, 2010.

[DC] PyPy Developer Community. RPython 2.6.0 documentation. `http://rpython.readthedocs.org` July 2015.

[DS84] L Peter Deutsch and Allan M Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM, 1984.

[DT14] Lukas Diekmann and Laurence Tratt. Eco: A language composition editor. In *Software Language Engineering*, pages 82–101. Springer International Publishing, 2014.

[ESEMEN09] Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 63–69. ACM, 2009.

[EWO04] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for simd architectures with alignment constraints. In *ACM SIGPLAN Notices*, volume 39, pages 82–93. ACM, 2004.

[FKÜ05] Franz Franchetti, Stefan Kral, and Jürgen Lorenz Christoph W. Überhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, Feb. 2005.

[FP02] Franz Franchetti and Markus Püschel. A SIMD vectorizing compiler for digital signal processing algorithms. In Bob Werner, editor, *16th International Parallel and Distributed Processing Symposium (IPDPS '02*

(IPPS & SPDP)), pages 20–21, Washington - Brussels - Tokyo, April 2002. IEEE.

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

[GPF06] Andreas Gal, Christian W Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153. ACM, 2006.

[IHWN11] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 246–256. IEEE Computer Society, 2011.

[KL00] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[LA00] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*, volume 35. ACM, 2000.

[LCF$^+$07] Piotr Lesnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andreas Ornstein, and Erven Rohou. Split compilation: an application to just-in-time vectorization. In *Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07)*, Brasov, Romania, 2007.

[Nai04] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[NDR$^+$11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor simd: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 151–160. IEEE Computer Society, 2011.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *ACM SIGPLAN Notices*, volume 41, pages 132–143. ACM, 2006.

[NZ06] Dorit Nuzman and Ayal Zaks. Autovectorization in gcc–two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158. Citeseer, 2006.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[Pal09] Mike Pall. LuaJit desgin note, 2009. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`, Sep. 2015.

[PKH03] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. *Pointer alignment analysis for processors with SIMD instructions*. na, 2003.

[PKH07] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Compiler optimizations for processors with SIMD instructions. *Software: Practice and Experience*, 37(1):93–113, 2007.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[RDN$^+$11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 35–44. ACM, 2011.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[RWP06] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for simd devices. In *ACM SIGPLAN Notices*, volume 41, pages 118–131. ACM, 2006.

[SGS10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[SHC05] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the international symposium on Code generation and optimization*, pages 165–175. IEEE Computer Society, 2005.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[TNC+09] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 327–337. IEEE, 2009.

[VRD11] Guido Van Rossum and Fred L Drake. *The Python language reference manual.* Network Theory Ltd., 2011.

[WB13] Christian Wimmer and Stefan Brunthaler. ZipPy on truffle: a fast and simple implementation of python. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 17–18. ACM, 2013.

[WEW05] Peng Wu, Alexandre E Eichenberger, and Amy Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 153–164. IEEE, 2005.

[WEWZ05] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 169–178, New York, NY, USA, 2005. ACM.

[WT92] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *Parallel and Distributed Systems, IEEE Transactions on*, 3(5):591–601, 1992.

[WW12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14. ACM, 2012.

[ZBS07] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. Yeti: a gradually Extensible Trace Interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93. ACM, 2007.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM, 1990.