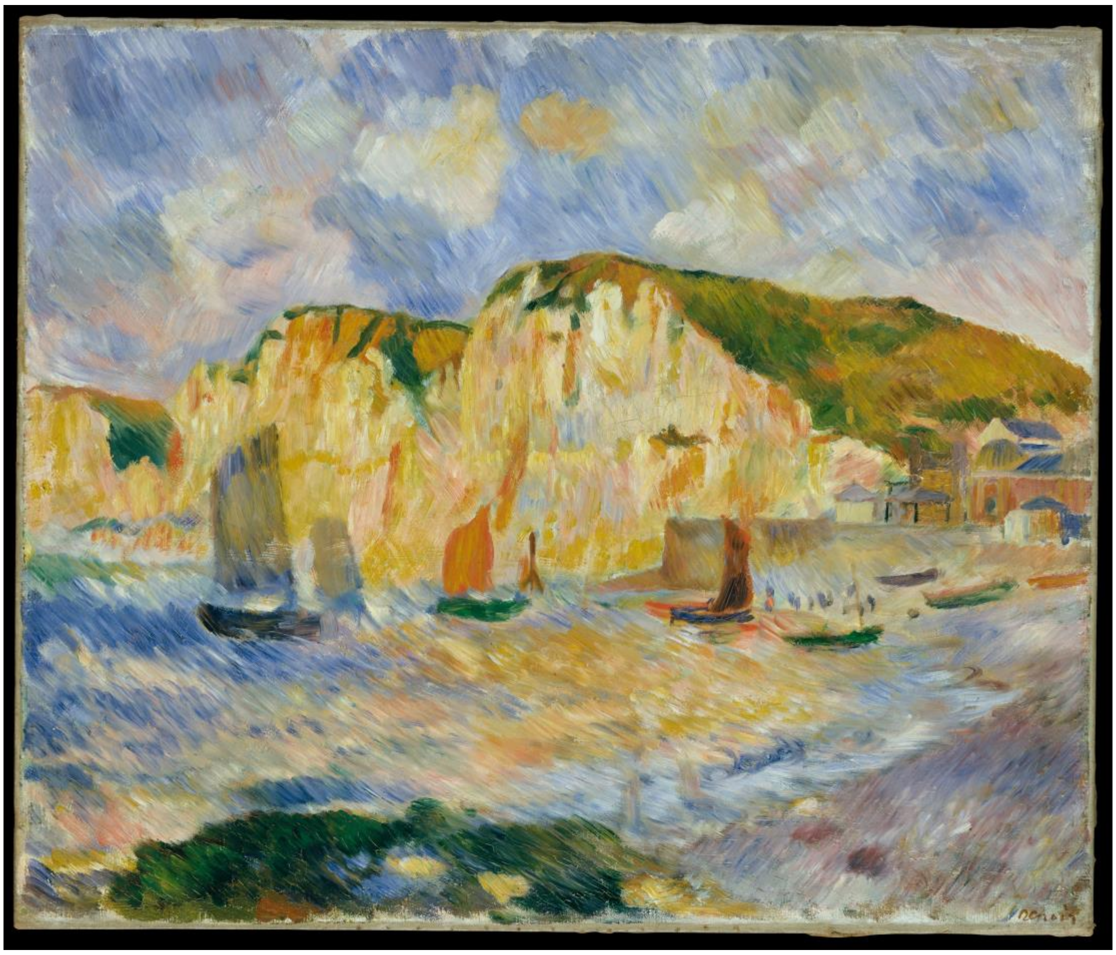
HELLENIC REPUBLIC
National and Kapodistrian University of Athens
Department of Informatics and Telecommunications

ABSTRACTS OF DOCTORAL DISSERTATIONS



Athens 2018
Volume 13

HELLENIC REPUBLIC
National and Kapodistrian
University of Athens

Department of Informatics and Telecommunications

ABSTRACTS OF DOCTORAL DISSERTATIONS

The Committee of Research and Development Activities

M. Koubarakis

E.S. Manolakos

T. Theoharis

ISSN: 1791-7948

Copyright © 2018

Volume 13

National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Panepistimiopolis, 15784 Athens, Greece

PREFACE

This volume includes extended abstracts of Doctoral Dissertations conducted in the Department of Informatics and Telecommunications, University of Athens, and completed from 1/2017 to 12/2017.

We publish this volume to demonstrate the breadth and quality of the original research performed by our Ph.D. students and faculty and to facilitate the dissemination of their innovative research results. We are happy to present the 13th yearly collection of this kind and expect this initiative to continue in the years to come. The submission of an extended abstract in English is required by all graduating doctoral students in our Department.

We would like to thank all graduates who contributed to this volume and hope that this was a positive experience for them. Finally, we would like to thank PhD candidate Nikos Bogdos for his help and attention to detail in putting together this volume.

The painting in the cover is called *Sea and Cliffs* (1885) by *Auguste Renoir (1841 - 1919)*.

The DiT Dept. Committee on Research and Development Activities

M. Koubarakis
E.S. Manolakos (publication coordinator)
T. Theoharis

Athens, June 2018

Table of Contents

Preface 3

Table of Contents 5

Doctoral Dissertations

George Balatsouras, *Recovering Structural Information for Better Static Analysis.* 7

Aggelos Biboudis, *Extensible and Efficient Streaming Libraries.* 19

Michail Bourbos, *Coupled semiconductor lasers and their applications in telecommunications and networks.* 31

Elias Konstantinidis, *A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling.* 43

Spiridoula Xenaki, *Advances in Possibilistic Clustering with Application to Hyperspectral Image Processing.* 55

Recovering Structural Information for Better Static Analysis

George Balatsouras[*]

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
gbalats@di.uoa.gr

Abstract. To reach a truly broad level of program understanding, static analysis techniques need to create an abstraction of memory that covers all possible executions. Such abstract models may quickly degenerate after losing essential structural information about the memory objects they describe, due to the use of specific programming idioms and language features, or because of practical analysis limitations. In many cases, some of the lost memory structure may be retrieved, though it requires complex inference that takes advantage of indirect uses of types. Such recovered structural information may, then, greatly benefit static analysis. This dissertation shows how we can recover structural information, first (i) in the context of C/C++, and next, in the context of higher-level languages without direct memory access, like Java, where we identify two primary causes of losing memory structure: (ii) the use of reflection, and (iii) analysis of partial programs. We show that, in all cases, the recovered structural information greatly benefits static analysis on the program.

Keywords: Pointer Analysis; Object-Oriented Programming; Type Hierarchy; Reflection

1 Introduction

The most promising and powerful of existing static analysis techniques rely on the creation of some *abstract memory model* of the program. What objects will the memory contain, at some state of execution? What will their structure be like? A faithful abstract representation of the actual memory is, however, a demanding task; its precision often decisive for the value of whatever the static analysis is aiming to eventually compute (be it the identification of complex bug patterns or the opportunities for effective optimizations).

Thesis.

There is *implicit structural information* in the program, about the memory it will allocate, that can improve the quality of the abstract memory

[*] Dissertation Advisor: Yannis Smaragdakis, Professor

model constructed by static analysis. This structural information is not readily available, but may be recovered via inference, primarily by tracking the use of types in the program.

We provide a number of techniques that recover such lost memory structure, in two different settings: (1) in C/C++ programs, as a typical case of low-level code with direct memory access, where the program's memory structure is often lost due to specific programming idioms and the inherent low-level nature of the language, and (2) in Java programs, where, despite the high-level nature of the language, structural information may be lost (a) for *partial programs* (i.e., libraries or any programs that lack some of their parts), which, in the form of Java Archives (JARs), constitute the main distributable code entity of this managed language, or (b) due to Java's *reflection* mechanism, which allows runtime inspection of classes, interfaces, fields and methods, and can be used to instantiate new objects, invoke methods, get/set field values, and so on, without exact static type information (e.g., the name of the method to be invoked can be created dynamically using plain string operations).

2 Structure-Sensitive Points-To Analysis for C and C++

Points-to analysis computes an abstract model of the memory that is used to answer the following query: *What can a pointer variable point-to, i.e., what can its value be when dereferenced during program execution?* This query serves as the cornerstone of many other static analyses aiming to enhance program understanding or assist in bug discovery (e.g., deadlock detection), by computing higher-level relations that derive from the computed points-to sets. In the literature, one can find a multitude of points-to analyses with varying degrees of precision and speed.

One of the most popular families of pointer analysis algorithms, *inclusion-based* analyses (or Andersen-style analyses [3]), originally targeted the C language, but has been extended over time and successfully applied to higher-level object-oriented languages, such as Java [6,7,21,25,29]. Surprisingly, precision-enhancing features that are common practice in the analysis of Java programs, such as field sensitivity or online call-graph construction are absent in many analyses of C/C++ [12,15,30,14,8,13].

In the case of field sensitivity, the reason behind its frequent omission when analyzing C is that it is much harder to implement correctly than in Java. As noted by Pearce et al. [24], the crucial difference is that, in C/C++, it is possible to have the address of a field taken, stored to some pointer, and then dereferenced later, at an arbitrarily distant program point. In contrast, Java does not permit taking the address of a field; one can only load or store to some field directly. Hence, `load/store` instructions in Java bytecode (or any equivalent IR) need an extra field specifier, whereas in C/C++ intermediate representations (e.g., LLVM bitcode) `load/store` requires only a single address operand. The precise field affected is not explicit, but only possibly computed by the analysis itself.

The effect of such difference in the underlying IRs, as far as pointer analysis is concerned, is far from trivial. In C, the computed points-to sets have an expanded domain, since now the analysis must be able to express that a variable `p` *at some offset* `i` may point-to another variable `q` *at some offset* `j`, with these offsets corresponding to either field components or array elements.

The best-documented approach on how to incorporate field sensitivity in a C/C++ points-to analysis is that of Pearce et al. [23,24]. The authors extend the constraint-graph of the analysis by adding (positive) weights to edges; the weights correspond to the respective field indices. For instance, the instruction "`q = &(p->f`$_\mathtt{i}$`)`" would be encoded as a constraint $q \supseteq p + i$. However, this approach does not take types into account. In fact, types are not even statically available at all allocation sites, since most standard C allocation routines are type-agnostic and return byte arrays that are cast to the correct type at a later point (e.g., `malloc()`, `realloc()`, `calloc()`). Thus, field i is represented with no regard to the type of its base object, even when this base object abstracts a number of concrete objects of different types. The lack of type information for abstract objects is a great source of imprecision, since it results in a prohibitive number of spurious points-to inferences.

We argue that type information is an essential part in increasing analysis precision, even when it is not readily available. The abstract object types should be rigorously recorded in all cases, especially when indexing fields, and used to filter the points-to sets. In this spirit, we present a *structure-sensitive* analysis for C/C++ that employs a number of techniques in this direction, aiming to retrieve high-level structure information for abstract objects in order to increase analysis precision:

1. First, the analysis records the type of an abstract object when this type is available at the allocation site. This is the case with stack allocations, global variables, and calls to C++'s `new()` heap allocation routine.

2. In cases where the type is not available (as in a call to `malloc()`), the analysis deviates from the allocation-site abstraction and creates multiple abstract objects per allocation site: one for every type that the object could have. Thus, each abstract object of type `T` now represents the set of all concrete objects of type `T` allocated at this site. To determine the possible types for a given allocation site, the analysis creates a special type-less object and records the cast instructions it flows to (i.e., the types it is cast to), using the existing points-to analysis. This is similar to the use-based *back-propagation* technique used in past work [17,19,27], in a completely different context—handling Java reflection.

3. The field components of abstract objects are represented as abstract objects themselves, as long as their type can be determined. That is, an abstract object `SO` of struct type `S` will trigger the creation of abstract object `SO.f`$_\mathtt{i}$, for each field `f`$_\mathtt{i}$ in `S`. (The aforementioned special objects trigger no such field component creation, since they are typeless.) Thus, the recursive cre-

ation of subobjects is bounded by the type system, which does not allow the declaration of types of infinite size.

4. Finally, the analysis treats array elements similarly to field components (i.e., by representing them as distinct abstract objects, if we can determine their type), as long as their respective indices statically appear in the source code. That is, an abstract object `AO` of array type `[T×N]` will trigger the creation of abstract object `AO[c]`, if the constant `c` is used to index into type `[T×N]`. The object `AO[*]` is also created, to account for indexing at unknown (variable) indices.

The last point offers some form of array-sensitivity as well and is crucial for analyzing C++ code, lowered to an intermediate representation such as LLVM bitcode, in which all the object-oriented features have been translated away. To be able to resolve virtual calls, an analysis must precisely reason about the exact v-table index that a variable may point to, and the method that such an index may itself point-to. That is, a precise analysis should not merge the points-to sets of distinct indices of v-tables.

We offer an implementation of our approach over the full LLVM bitcode intermediate language, in the form of a new static analysis tool, `cclyzer`[1]. We show that our approach yields much higher precision than past analyses, allowing accurate distinctions between subobjects, v-table entries, array components, and more. Especially for C++ programs, this precision is invaluable for a realistic analysis. Compared to the state-of-the-art past approach, our techniques exhibit substantially better precision along multiple metrics and realistic benchmarks (e.g., 40+% more variables with a single points-to target).

3 More Sound Static Handling of Java Reflection

Moving to higher-level languages, like Java, we note that essential structural information is often lost in Java programs too, yet for different reasons. A source of analysis imprecision, especially in determining the types of abstract objects constructed by the analysis, lies in the use of Java's reflection mechanism: the ability to inspect and dynamically retrieve classes, methods, attributes, etc. at runtime.

By using the Reflection API, Java programs can encompass dynamic behavior. However, statically reasoning about the behavior of software that uses reflection can be especially cumbersome. Unfortunately, reflection is ubiquitous in large Java programs. When a Java program accesses a class by supplying its name as a run-time string, via the `Class.forName` library call, the static analysis has very few available courses of action: It needs to either conservatively over-approximate (e.g., assume that *any* class can be accessed, possibly limiting the set later, after the returned object is used), or to perform a string analysis that will allow it to infer the contents of the `forName` string argument. Both

[1] `cclyzer` is publicly available at `https://github.com/plast-lab/cclyzer`

options can be detrimental to the scalability of the analysis: the conservative over-approximation may never become constrained enough by further instructions to be feasible in practice; precise string analysis is impractical for programs of realistic size. It is telling that *no practical Java program analysis framework in existence handles reflection soundly* [18], although other language features are modeled soundly.[2]

Full soundness is not practically achievable, but it can still be approximated for the well-behaved reflection patterns encountered in regular, non-adversarial programs. Therefore, it makes sense to treat soundness as a continuous quantity: something to improve on, even though we cannot perfectly reach. To avoid confusion, we use the term *empirical soundness* for the quantification of how much of the dynamic behavior the static analysis covers. Computable metrics of empirical soundness can help quantify how close an analysis is to the fully sound result. Based on such metrics, one can make comparisons (e.g., "more sound") to describe soundness improvements.

The second challenge of handling reflection in a static analysis is *scalability*. The online documentation of the IBM WALA library [10] concisely summarizes the current state of the practice, for *points-to analysis* in the Java setting.

> *Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings,* WALA*'s pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.*

The same caveats routinely appear in the research literature. Multiple published points-to analysis papers analyze well-known benchmarks with reflection disabled [28,16,1,2].

A representative quote [28] illustrates:

> *Hsqldb and jython could not be analyzed with reflection analysis enabled [...] —hsqldb cannot even be analyzed context-insensitively and jython cannot even be analyzed with the 1obj analysis. This is due to vast imprecision introduced when reflection methods are not filtered in any way by constant strings (for classes, fields, or methods) and the analysis infers a large number of reflection objects to flow to several variables. [...] For these two applications, our analysis has reflection reasoning disabled. Since hsqldb in the DaCapo benchmark code has its main functionality called via reflection, we had to configure its entry point manually.*

We describe an approach for handling reflection with improved empirical soundness (as measured against prior approaches and dynamic information), again, in the context of a points-to analysis. Our approach is based on the combination of string-flow and points-to analysis from past literature augmented with (a) substring analysis and modeling of partial string flow through string builder classes;

[2] In our context, *sound* = over-approximate, i.e., guaranteeing that all possible behaviors of reflection operations are modeled.

(b) new techniques for analyzing reflective entities based on information available at their use-sites. In experimental comparisons with prior approaches, we demonstrate a combination of both improved soundness (recovering the majority of missing call-graph edges) and increased performance. Our approach requires no manual configuration and achieves significantly higher empirical soundness without sacrificing scalability, for realistic benchmarks and libraries (DaCapo Bach and Java 7).

In experimental comparisons with the recent Elf system [17] (itself improving over the reflection analysis of the Doop framework [7]), our algorithm discovers most of the call-graph edges missing (relative to a dynamic analysis) from Elf's reflection analysis. This improvement in empirical soundness is accompanied by *increased* performance relative to Elf, demonstrating that near-sound handling of reflection is often practically possible. Concretely, our work for reflection:

· introduces key techniques in static reflection handling that contribute greatly to empirical soundness. The techniques generalize past work from an intra-procedural to an inter-procedural setting and combine it with a string analysis;
· shows how scalability can be addressed with appropriate tuning of the above generalized techniques;
· thoroughly quantifies the empirical soundness of a static points-to analysis, compared to past approaches and to a dynamic analysis;
· is implemented and evaluated on top of an existing open framework (Doop [7]).

4 Class Hierarchy Complementation for Java

Whole-program static analysis is essential for clients that require high-precision and a deeper understanding of program behavior. Modern applications of program analysis, such as large scale refactoring tools [9], race and deadlock detectors [22], and security vulnerability detectors [20,11], are virtually inconceivable without whole-program analysis.

For whole-program analysis to become truly practical, however, it needs to overcome several real-world challenges. One of the somewhat surprising real-world observations is that whole-program analysis requires the availability of much more than the "whole program". The analysis needs an overapproximation of what constitutes the program. Furthermore, this overapproximation is not merely what the analysis computes to be the "whole program" after it has completed executing. Instead, the overapproximation needs to be as conservative as required by any intermediate step of the analysis, which has not yet been able to tell, for instance, that some method is never called.

Consider the example of trying to analyze a program P that uses a third-party library L. Program P will likely only need small parts of L. However, other, entirely separate, parts of L may make use of a second library, L'. It is typically not possible to analyze P with a whole program analysis framework without also supplying the code not just for L but also for L', which is an unreasonable burden. In modern languages and runtime systems, L' is usually not necessary in order to either compile P or run it under any input. The problem is exacerbated in the current era of large-scale library reuse. In fact, it is often the case that the user is not even aware of the existence of L' until trying to analyze P.

Our research consists precisely of addressing such need in full generality. *Given a set of Java class and interface definitions, in bytecode form, we compute a "program complement", i.e., skeletal versions of any referenced missing classes and interfaces so that the combined result constitutes verifiable Java bytecode.*

To see why the problem has interesting depth and complexity, consider a simple fragment of Java bytecode and the constraints it induces. Our convention here is that single-letter class names at the lower end of the alphabet (`A`, `B`, ...) correspond to known types, while class names at the high end of the alphabet (`X`, `Y`, `Z`) denote phantom types. We present bytecode in a slightly condensed form, to make clear what method names or type names are referenced in every instruction.

```
public void foo(X, Y)
0: aload_2     // load on stack 2nd argument (of type Y)
1: aload_1     // load on stack 1st argument (of type X)
2: invokevirtual X.bar:(LA;)LZ; // method 'Z bar(A)' in X
3: invokevirtual B.baz:()V;     // method 'void baz()' in B
 ...
```

Although the above fragment is merely four bytecode instructions long, it induces several interesting constraints for our phantom types `X`, `Y`, and `Z`:

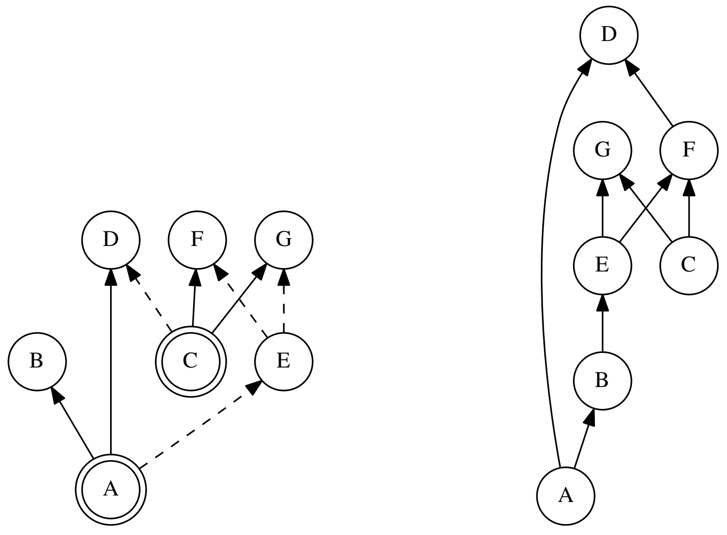
– `X` has to support a method `bar` accepting an argument of type `A` and returning a value of type `Z`.
– `Y` has to be a subtype of `A`, since an actual argument of declared type `Y` is passed to `bar`, which has a formal parameter of type `A`. This constraint also means that if `A` is known to be a class (and not an interface) then `Y` is also a class.
– `Z` has to be a subtype of `B`, since a method of `B` is invoked on an object of declared type `Z` (returned on top of the stack by the earlier invocation).

Our goal is to satisfy all such constraints and generate definitions of phantom types `X`, `Y`, and `Z` that are compatible with the bytecode that is available to the tool (i.e., exists in known classes). Compatibility with existing bytecode is defined as satisfying the requirements of the Java verifier, which concern type well-formedness.

Note that such definitions will contain essential parts of missing structural information for the phantom types: method and field members, as well as supertypes. Any subsequent static analysis that will operate on the types produced by complementation will create abstract objects that are much closer, in structure, to reality.

Of these constraints, the hardest to satisfy are those involving subtyping. Constraints on members (e.g., `X` has to contain a "`Z bar(A)`") are easy to satisfy by just adding type-correct dummy members to the generated classes. This means that the core of the general program complementation problem is solving the *class hierarchy complementation problem*: given a partial type hierarchy and a set of subtyping constraints, compute a complete type hierarchy that satisfies the subtyping constraints *without* changing the direct parents of known types.

Solving the hierarchy complementation problem, constitutes the main novelty of our approach. The problem appears to be fundamental, and even of a certain interest in purely graph-theoretic terms. For a representative special case, consider an object-oriented language with multiple inheritance (or, equivalently, an interface-only hierarchy in Java or C#). A partial hierarchy, augmented with constraints, can be represented as a graph, as shown in Figure 1a. The known part of the hierarchy is shown as double circles and solid edges. Unknown (i.e., missing) classes are shown as single circles. Dashed edges represent subtyping constraints, i.e., indirect subtyping relations that have to hold in the resulting hierarchy. In graph-theoretic terms, a dashed edge means that there is a path in the solution between the two endpoints. For instance, the dashed edge from C to D in Figure 1a means that the unknown part of the class hierarchy has a path from C to D. This path cannot be a direct edge from C to D, however: C is a known class, so the set of its supertypes is fixed.



(a) Constraint Graph (b) Solution

Fig. 1: Example of constraints in a multiple inheritance setting. Double-circles signify known classes, single circles signify unknown classes. Solid edges ("known edges") signify direct subtyping, dashed edges signify transitive subtyping.

In order to solve the above problem instance, we need to compute a directed acyclic graph (DAG) over the same nodes,[3] so that it preserves all known nodes and edges, and adds edges *only to unknown nodes* so that all dashed-edge constraints are satisfied. That is, the solution will not contain dashed edges (indirect subtyping relationships), but every dashed edge in the input will have a matching directed path in the solution graph. Figure 1b shows one such possible solution. As can be seen, solving the constraints (or determining that they are unsatisfiable) is not trivial. In this example, any solution has to include an edge from B to E, for reasons that are not immediately apparent. Accordingly, if we change the input of Figure 1a to include an edge from E to B, then the constraints are not satisfiable—any attempted solution introduces a cycle. The essence of the algorithmic difficulty of the problem (compared to, say, a simple topological sort) is that we cannot add extra direct parents to known classes A and C—any subtyping constraints over these types have to be satisfied via existing parent types. This corresponds directly to our high-level program requirement: we want to compute definitions for the missing types only, without changing existing code. For a language with single inheritance, the problem is similar, with one difference: the solution needs to be a tree instead of a DAG. (Of course, the input in Figure 1a already violates the tree property since it contains known nodes with multiple known parents.)

We provide algorithms to solve the hierarchy complementation problem in the single inheritance and multiple inheritance settings. We also show that the problem in a language such as Java, with single inheritance but multiple subtyping and distinguished class vs. interface types, can be decomposed into separate single- and multiple-subtyping instances. We implement our algorithms in a tool, JPhantom,[4] which complements partial Java bytecode programs so that the result is guaranteed to satisfy the Java verifier requirements. In a sense, JPhantom aims to recover structural information for phantom classes, via inference, by tracking their use in existing code. JPhantom is highly scalable and runs in mere seconds even for large input applications and complex constraints (with a maximum of 14s for a 19MB binary).

5 Conclusions

To summarize, we advocate that there are many opportunities in recovering implicit structural information about memory that can improve static analysis of programs, but require complex inference that takes advantage of indirect uses of types. We have examined three different scenarios to test and evaluate our thesis, regarding generic C/C++ programs, and Java programs that either use reflection or are missing parts of their code. In all cases, we where able to improve static analysis, by recovering memory structure that was not previously evident.

[3] Inventing extra nodes does not contribute to a solution in this problem.

[4] JPhantom is available online at `https://github.com/gbalats/jphantom`

6 Publications

The contents of this doctoral dissertation are based on the following published papers:

– *Structure-Sensitive Points-To Analysis for C and C++* [5]
– *More Sound Static Handling of Java Reflection* [27]
– *Class Hierarchy Complementation: Soundly Completing a Partial Type Graph* [4]
– *Pointer Analysis* [26]

References

1. Ali, K., Lhoták, O.: Application-only call graph construction. In: Proc. of the 26th European Conf. on Object-Oriented Programming. pp. 688–712. ECOOP '12, Springer (2012)
2. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Proc. of the 27th European Conf. on Object-Oriented Programming. pp. 378–400. ECOOP '13, Springer (2013)
3. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (May 1994)
4. Balatsouras, G., Smaragdakis, Y.: Class hierarchy complementation: Soundly completing a partial type graph. In: Proc. of the 28th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 515–532. OOPSLA '13, ACM, New York, NY, USA (2013)
5. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for C and C++. In: Proc. of the 23rd International Symp. on Static Analysis. SAS '16, Springer (2016)
6. Berndl, M., Lhoták, O., Qian, F., Hendren, L.J., Umanee, N.: Points-to analysis using BDDs. In: Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 103–114. PLDI '03, ACM, New York, NY, USA (2003)
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. OOPSLA '09, ACM, New York, NY, USA (2009)
8. Das, M.: Unification-based pointer analysis with directional assignments. In: Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 35–46. PLDI '00, ACM, New York, NY, USA (2000)
9. Dig, D.: A refactoring approach to parallelism. IEEE Software 28(1), 17–22 (2011)
10. Fink, S.J., et al.: WALA UserGuide: PointerAnalysis. `http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis`
11. Guarnieri, S., Livshits, B.: GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In: Proc. of the 18th USENIX Security Symposium. pp. 151–168. SSYM' 09, USENIX Association, Berkeley, CA, USA (2009), `http://dl.acm.org/citation.cfm?id=1855768.1855778`
12. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 290–299. PLDI '07, ACM, New York, NY, USA (2007)

13. Hardekopf, B., Lin, C.: Exploiting pointer and location equivalence to optimize pointer analysis. In: Proc. of the 14th International Symp. on Static Analysis. pp. 265–280. SAS '07, Springer (2007)
14. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 254–263. PLDI '01, ACM, New York, NY, USA (2001)
15. Hind, M., Burke, M.G., Carini, P.R., Choi, J.: Interprocedural pointer alias analysis. ACM Trans. on Programming Languages and Systems 21(4), 848–894 (1999)
16. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. In: Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation. PLDI '13, ACM, New York, NY, USA (2013)
17. Li, Y., Tan, T., Sui, Y., Xue, J.: Self-inferencing reflection resolution for Java. In: Proc. of the 28th European Conf. on Object-Oriented Programming. pp. 27–53. ECOOP '14, Springer (2014)
18. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Communications of the ACM 58(2), 44–46 (Jan 2015)
19. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 139–160. APLAS '05, Springer (2005)
20. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering. pp. 499–509. FSE '13, ACM (2013)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: Proc. of the 2002 International Symp. on Software Testing and Analysis. pp. 1–11. ISSTA '02, ACM, New York, NY, USA (2002)
22. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 308–319. PLDI '06, ACM, New York, NY, USA (2006)
23. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis for C. In: Proc. of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 37–42. PASTE '04, ACM, New York, NY, USA (2004)
24. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis of C. ACM Trans. on Programming Languages and Systems 30(1) (2007)
25. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Proc. of the 16th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 43–55. OOPSLA '01, ACM, New York, NY, USA (2001)
26. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Foundations and Trends® in Programming Languages 2(1), 1–69 (2015), `http://dx.doi.org/10.1561/2500000014`
27. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of Java reflection. In: Proc. of the 13th Asian Symp. on Programming Languages and Systems. pp. 485–503. APLAS '15, Springer (2015)
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proc. of the 38th ACM SIGPLAN-SIGACT Symp.

on Principles of Programming Languages. pp. 17–30. POPL '11, ACM, New York, NY, USA (2011)
29. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for Java programs. In: Proc. of the 14th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications. pp. 187–206. OOPSLA '99, ACM, New York, NY, USA (1999)
30. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. pp. 197–208. POPL '08, ACM, New York, NY, USA (2008)

Extensible and Efficient Streaming Libraries

Aggelos Biboudis*

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
biboudis@di.uoa.gr

Abstract. Stream processing is mainstream (again): Widely-used stream libraries are now available for virtually all modern OO and functional languages, from Java to C# to Scala to OCaml to Haskell. Yet expressivity and performance are still lacking. This dissertation identifies the key high-level differences between various implementations, observes that future use cases are tied with past design decisions, and shows simple abstraction mechanisms are not sufficient. Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance? We present a twofold modularization of streams. To begin with, we untangle streams from the definition of their syntax and semantics and afterwards we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.
Our contributions are preceded by a performance assessment, of the current state-of-the-art of streaming libraries. Subsequently, we first propose a mechanism to enhance the maintainability of streams, supporting a high-level of extensibility. We treat streams as a domain-specific language and we design and implement **StreamAlg**, a library that has the ability to accept new operators and semantics á la carte. Next, we port the library design we used for streams to Java itself, with a lightweight tool named **Recaf**. We show how to create dialects in Java, override its semantics, support new syntactic elements and much more. Among many examples and case studies we build an extension of Java with a keyword that enables us to construct streams similar to C#. The culmination of our work is a library design, **Strymonas**, for very efficient streams while preserving their high-level nature. It explicitly avoids the reliance on black-box optimizers and "sufficiently-smart" compilers, offering highest, guaranteed and portable performance. Our approach relies on high-level concepts that are then readily mapped into an implementation.

Keywords: Code generation, domain-specific languages, multi-stage programming, optimization, stream fusion, streams

1 Introduction

Programming languages have started shifting away from the sequential programming model that the von Neumann architecture so vigorously imposed [2]. Instead

* Dissertation Advisor: Yannis Smaragdakis, Professor

of thinking in terms of *commands* and static *storage*, modern programming needs often encourage thinking in terms of processes and transformations over *flows* of data. That transition happened over several decades of research and development in programming languages, systems, and computer architectures. *Streaming* functionality is a prominent representative of this trend. Casually speaking, in computer science, a stream is a sequence of elements that can be piped through a series of transformation steps. A streaming library is a software library to manipulate streams. All streaming libraries seem to fulfill similar goals; however, their vastly different characteristics make them one of the most fascinating areas of software construction.

This dissertation investigates the modern design decisions behind the streaming libraries that are used in general-purpose programming. We identify the key high-level differences between various implementations and observe that future use cases are tied with past design decisions and simple abstraction mechanisms are not sufficient. Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance? We present a twofold modularization of streams. Firstly, we untangle streams from the definition of their syntax and semantics, and secondly, we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.

Nowadays, streaming libraries let us model algorithms as if data were in motion and not stationary: stock ticks, tweets, sales, products, inventory and real-time analytics are only some of the examples that generate petabytes of information available to data scientists. In terms of conceptual modeling, a stream corresponds to a pipe transporting gas or liquids over long distances. Materials are being processed in location A where an activity f takes place. After processing ends, each element is put in the pipe and is transferred to another location B, where f' takes place. The pipe represents the flow of data and the activities f and f' represent transformations on each element of the stream. We have only declared *what* activities take place and not *how* each transformation works. "Stream processing lets us model systems that have state without ever using assignment or mutable data" per the authors of *Structure and Interpretation of Computer Programs* [1].

A streaming library is typically offered with a set of operators to create streams, transform and consume them into scalar or other kinds of data structures, as shown in Figure 1. Its distinguishing feature in relation to simple collection APIs is that intermediate transformations are performed *on-demand*, thus they do not perform more computation than needed. Producer operators can be either backed by an in-memory data structure or not. `of_arr` creates a stream out of an array and `unfold` builds a (possibly unbounded) stream from a seed value (it unfolds a whole stream from a single value). Next are operators that transform a stream. A stream can be transformed either in a linear or a non-linear way. `map` applies a function f to each element of the input stream and returns a transformed stream. The number of elements on the input streams is equal to the number on the transformed stream. This is where linearity comes from. On the contrary,

```
// Producers of finite
val of_arr   : 'a array → 'a stream

// Producers of possibly infinite
val unfold   : ('state  → ('a * 'state) option) → 'state → 'a stream

// Transformers of linear nature
val map      : ('a → 'b) → 'a stream → 'b stream

// Transformers of non-linear nature
val filter   : ('a → bool) → 'a stream → 'a stream
val take     : int → 'a stream → 'a stream
val flat_map : ('a → 'b stream) → 'a stream → 'b stream

// Transformers of parallel loops
val zip_with : ('a → 'b  → 'c) → 'a stream → 'b stream  → 'c stream

// Consumers
val fold     : ('state → 'a → 'state) → 'state → 'a stream → 'state
```

Fig. 1: Stream Operators

`filter` applies a predicate to the input stream, again element-wise and unless the predicate is satisfied, the element does not appear on the output stream. Other operators like `take` sub-range the input stream based on a counter value. `flat_map` applies a function to each element; the results of the function application are concatenated to form the output stream, which can be a stream of zero, one or more elements. `zip_with` merges two streams according to a zipping function, applied element-wise over two streams. As expected, `zip_with` can have variations on the number of input streams as well as a default behavior like zipping two elements into a pair (called simply `zip`). Finally, we have consumers, like `fold` which apply a binary function, combining all elements of the stream. `fold` is a standard recursion operator for processing lists and can be used to fold a stream (like folding a piece of paper) into something else: `sum`, `product`, `max`, `min`, `count`, boolean operations like disjunction `or` and conjunction `and`, `concat` are only some functions that can be implemented in terms of `fold`.[1]

We present the same pipeline in two language (Figures 2 and 3). Both pipelines calculate the sum of squared elements of an array.

[1] In fact, fold is highly powerful and standard operators like `map` and `filter` can also be implemented in terms of it.

```
def sumOfSquares(arr : Array[Double]) : Double = {
  val sum : Double = arr.view
     .map(a_i => a_i * a_i)
     .sum
  sum
}
```

Fig. 2: Sum of squares in Scala

```
public double sumOfSquares(double[] arr) {
  double sum = DoubleStream.of(arr)
     .map(a_i -> a_i * a_i)
     .sum();
  return sum;
}
```

Fig. 3: Sum of squares in Java 8

2 Two Modern Needs: Extensibility & Performance

The rationale behind streams for general-purpose programming is that they can be used for fast data processing by providing a minimal and easy-to-use abstraction. However, the design decisions behind them tie the implementation with future use cases. Consider the mainstream, VM-based, multi-paradigm programming languages C# (through the `System.Collections.IEnumerable` interface) and Java (through the `java.util.stream` interface), which offer vastly different designs for streams. While the first offers a `zip` operator, the second does not, sacrificing the functionality in favor of performance. Another example is that Java 8 Streams, due to their internal structure, following a *push-based* design, significantly outperform C#, following a *pull-based* design, in a number of occasions. On the flipside, C# guarantees laziness in more cases, often permitting higher memory efficiency.

Two key observations motivate our study. The first is that streams need not be tightly coupled to either their implementation or the range of operators they support. The user can freely change the underlying semantics for any reason. To achieve this we propose a new design for streams, **StreamAlg**, and we view the API of streaming libraries as a domain specific language (DSL). Using that perspective we can study both their syntactic and their semantic elements. In order to modularize streams on both, we isolate the functionally-inspired API that all stream APIs share, and we propose an extensible design. This will give users the opportunity to use different flavors of streams at will. One flavor could boost performance, another could trace execution steps, yet another could be the combination of the two!

Next, we apply the same design on the programming language Java. We propose **Recaf**, a compiler that liberates both the syntax and the semantics of Java and offers the same level of extensibility at the language level. Using that compiler we are able to create extensions for constructs that do not exist in Java such as a `yield` keyword to implement iterators and subsequently a stream library that follows the C# architecture in Java.

The second observation is that modern libraries rely either on extensible compilers or on a “sufficiently-smart” dynamic compiler to generate efficient machine-level code—as if the original source code had been loop-based, hand-written code with state, mutation and ... human intuition. In the first case, for example, Haskell provides rewrite rules on the `GHC.Base` and `GHC.List` modules to perform elimination of intermediate data structures. The rules are applied at compile time [7,19]. Library authors following this strategy usually maintain two code bases (possibly in the same compilation unit; yet programming two different things): a) the library itself (following a certain pattern), and b) the optimizations in the form of rewriting rules. For the second case, of a “sufficiently-smart” dynamic compiler, the underlying VM technologies are exceptional pieces of engineering and tremendously complex, like the Java Hotspot Server Compiler [14]. However, sometimes it is difficult to predict their behavior. For example the point that a streams is used may fail to inline (unfold its body) so the quality of the expected loop can be very poor.[2] In this dissertation we view these optimizations as domain–specific entirely and implement them explicitly in the stream library itself. We propose **Strymonas**, a library that embodies the level of separability described above to streams. We implement Strymonas in both OCaml and Scala.

3 Introducing the StreamAlg design

The new design we propose offers streaming libraries *à la carte* to maximize extensibility. Our approach requires no language changes, and only leverages features found across all languages examined—i.e., standard parametric polymorphism (generics). We argue for the benefits of this design in terms of extensibility and low adoption barrier (i.e., use of only standard language features), all without sacrificing performance. Additionally, we demonstrate extensibility and provide several alternative semantics for streaming pipelines, all in an actual, publicly available implementation. Finally, we provide an example of the use of object algebras in a real-world, performance-critical setting.

Underlying our architecture is the object algebra construction of Oliveira and Cook [12] and Oliveira et al. [13]. This is combined with a library design that dissociates the push or pull nature of iteration from the operators themselves,

[2] A quote by John Rose discussing two design strategies for Java 8 Streams on the [hotspot-compiler-dev] mailing list: “HotSpot are less good at internal iterators. If the original point of the user request fails to inline all the way into the internal looping part of the algorithm (a hidden ”for” loop), the quality of the loop will be very poor. ”—`https://web.archive.org/web/20170322141224/http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html`

analogously to the recent “defunctionalization of push arrays” approach in the context of Haskell [20].

In StreamAlg, a pipeline, shown earlier, gets inverted and parameterized by an `alg` object, which designates the intended semantics. For instance, a plain Java-streams-like evaluation would be written as in Figure 4.

```
PushFactory alg = new PushFactory();
int sum = Id.prj(
            alg.sum(
              alg.map(x → x * x,
                 alg.source(v))))).value;
```

Fig. 4: Example pipeline with push-based semantics

(The `Id.prj` and `value` elements in Figure 4 are part of a standard pattern for simulating higher-kinded polymorphism with plain generics. They can be ignored for the purposes of understanding our architecture.)

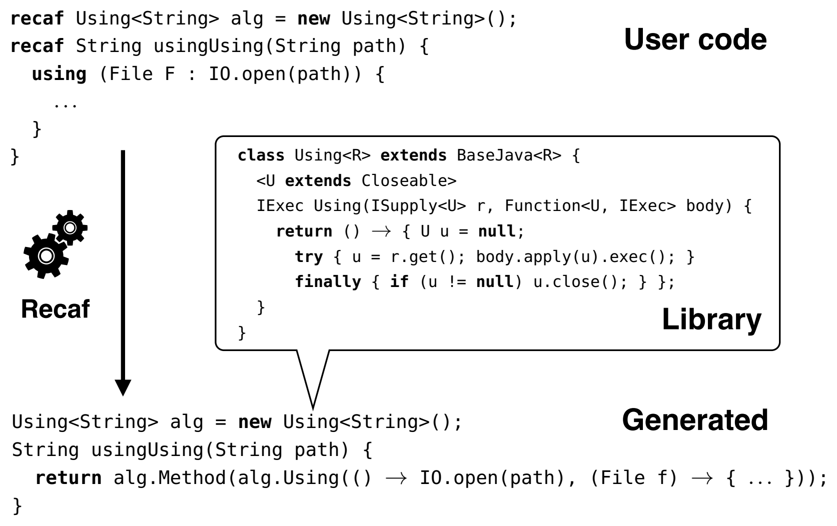
Although the code in Figure 4 is slightly longer than pipelines we showed earlier, its elements are highly stylized. The user can adapt the code to other pipelines with trivial effort, comparable to that of the original code fragment in Java 8 streams. Most importantly, if the user desired a different interpretation of the pipeline, the only necessary change is to the first line of the example. An interpretation that has pull semantics and fuses operators together only requires a new definition of `alg`:

```
FusedPullFactory alg = new FusedPullFactory();
... // same as earlier
```

Fig. 5: Declaration of an interpretation

Such new semantics can be defined externally to the library itself. Adding `FusedPullFactory` requires no changes to the original library code, allowing for semantics that the library designer had not foreseen.

This highly extensible design comes at no cost to performance. The new architecture introduces no extra indirection and does not prevent the JIT compiler from performing any optimization. This is remarkable, since current Java 8 streams are designed with performance in mind (cf. the earlier push-style semantics). As we show, StreamAlg matches or exceeds the performance of Java 8 streams.



```
recaf Using<String> alg = new Using<String>();
recaf String usingUsing(String path) {
  using (File F : IO.open(path)) {
    ...
  }
}
```

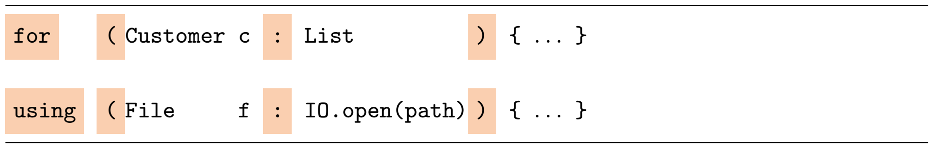
```
class Using<R> extends BaseJava<R> {
  <U extends Closeable>
  IExec Using(ISupply<U> r, Function<U, IExec> body) {
    return () → { U u = null;
      try { u = r.get(); body.apply(u).exec(); }
      finally { if (u != null) u.close(); } };
  }
}
```

```
Using<String> alg = new Using<String>();
String usingUsing(String path) {
  return alg.Method(alg.Using(() → IO.open(path), (File f) → { ... }));
}
```

Fig. 6: High level overview of Recaf

4 Introducing the Recaf library

Figure 6 gives a bird's eye overview of Recaf. It shows how a new language extension extension is used and implemented with Recaf. The new extension offer the same functionality with `try-with-resources` in Java and is called `using`. The top shows a snippet of code illustrating how the programmer would use a Recaf extension, in this case consisting of the `using` construct. The programmer writes an ordinary method, decorated with the `recaf` modifier to trigger the source-to-source transformation. To provide the custom semantics, the user also declares a `recaf` variable, in scope of the `recaf` method. In this case, an object with static type of `Using<String>` is defined (`alg` in this example).



```
for    ( Customer c : List          ) { ... }

using  ( File     f : IO.open(path) ) { ... }
```

Fig. 7: Recaf matching fragments over the concrete syntax

The downward arrow indicates Recaf's source-to-source transformation. Recaf **detects** that the new keyword relies on the `for`-each statement syntactically. An enhanced `for`-loop, in vanilla Java, omits explicit looping variables by operating over objects of type `*Iterable`. The new keyword, `using`, relies on the same pattern and the two uses are shown below for comparison (the highlighted parts in Figure 7 show the fragments over the concrete syntax, that the Recaf compiler matches to detect the pattern).

Recaf, after detecting the concrete syntax of the pattern, virtualizes the compilation unit at the method level by transforming the code fragment that includes the extension to the plain Java code at the bottom.

Each statement in the user code is transformed into calls on the `alg` object. The `using` construct itself is mapped to the `Using` method. The `Using` class, shown in the call-out, defines the semantics for `using`. It takes two parameters: one of type `ISupply`, a lambda that takes no parameters and supplies a value (the value on the right of the semicolon) and one function of type `Function<U, IExec>` that represents the code inside the block of `using`, as a function, parameterized by a value of type `U` (one the left of the semicolon and of type `File` in this example). It extends a class (`BaseJava`) capturing the ordinary semantics of Java, and defines a single method, also called `Using`. This particular `Using` method defines the semantics of the `using` construct as a kind of interpreter, of type `IExec`.

5 Introducing the Strymonas library

We next present **Strymonas**: a streaming library design that offers both high expressiveness and *guaranteed*, highest performance. First, we support the full range of streaming operators (a.k.a. stream *transformers* or *operators*) from past libraries: not just `map` and `filter` but also sub-ranging (`take`), nesting (`flat_map`—a.k.a. `concatMap`) and parallel (`zip_with`) stream processing. All operators are freely composable: e.g., `zip_with` and `flat_map` can be used together, repeatedly, with finite or infinite streams. Our novel stream representation captures the essence of stream processing for virtually all operators examined in past literature.

Second, our stream representation allows eliminating the abstraction overhead altogether, for the full set of stream operators. We perform *stream fusion* and other aggressive optimizations. The generated code contains no extra heap allocations in the main loop. By not generating tuples or other objects, we avoid the overhead of dynamic object construction and pattern-matching, and also the hidden, often significant overhead of memory pressure and boxing of primitive types as in Java 8 (using the generic types and not the hand-specialized) and in Scala. The result not merely approaches but attains the performance of hand-optimized code, from the simplest to the most complex cases, up to *well over* the complexity point where hand-written code becomes infeasible. Although the library operators are purely functional and freely composable, the actual running stream code is loop-based, highly tangled and imperative.

Our technique relies on staging, a form of metaprogramming, to achieve guaranteed stream fusion. This is in contrast to past use of source-to-source transformations of functional languages [8], of AST run-time rewriting [11,15], compile-time macros [17] or Haskell GHC Rules [16,6] to express domain-specific streaming optimizations.

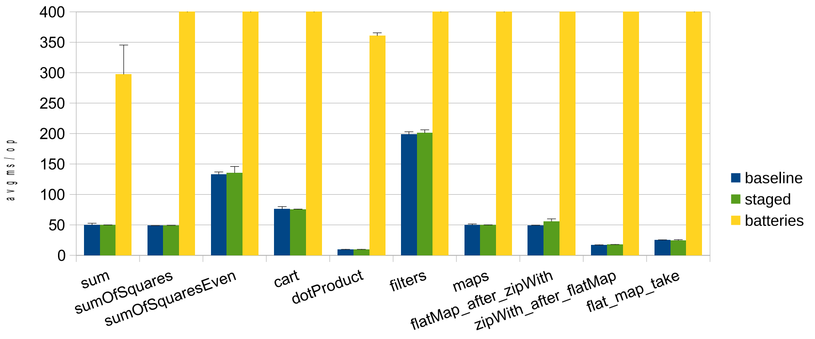


Fig. 8: OCaml microbenchmarks in msec / iteration (avg. of 30, with mean-error bars shown). “Staged” is our library (Strymonas). The figure is truncated: OCaml batteries take more than 60sec (per iteration!) for some complex benchmarks.

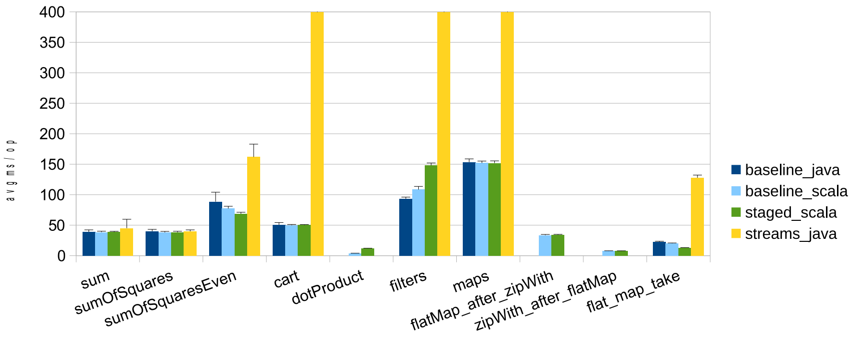


Fig. 9: JVM microbenchmarks (both Java and Scala) in msec / iteration (avg. of 30, with mean-error bars shown). “Staged_scala” is our library (Strymonas). The figure is truncated.

Rather than relying on an optimizer to eliminate artifacts of stream composition, we do not introduce the artifacts in the first place. Our library transforms highly abstract stream pipelines to code fragments that use the most suitable imperative features of the host language. The appeal of staging is its certainty and guarantees. Unlike the aforementioned techniques, staging also ensures that the generated code is well-typed and well-scoped, by construction. Our work describes a general approach, and not just a single library design. To demonstrate the generality of the principles, we implemented two library versions in diverse settings. The first is an OCaml library, staged with BER MetaOCaml [9]. The second is a Scala library (also usable by client code in Java and other JVM languages), staged with Lightweight Modular Staging (LMS) [18].

We evaluate Strymonas on a suite of benchmarks (Figures 8 and 9), comparing with hand-written code as well as with other stream libraries (including Java 8 Streams). Our staged implementation is up to more than two orders-of-magnitude faster than standard Java/Scala/OCaml stream libraries, matching the performance of hand-optimized loops. (Indeed, we occasionally had to improve hand-written baseline code, because it was slower than the library.)

Thus, our contributions are: (i) the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit the full elimination of abstraction overhead. The main principle is a novel representation of streams that captures rate properties of stream transformers and the form of termination conditions, while separating and abstracting components of the entire stream state. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration. (ii) The implementation of the design in terms of two distinct library versions for different languages and staging methods: OCaml/MetaOCaml and Scala/JVM/LMS.

6 Conclusions

Summarizing, we improve streams in terms of extensibility and performance, and with the mechanisms we present, we enhance them without breaking their high level structure. In this dissertation we treat interpretations and optimizations as pluggable components and we advocate that domain-specific optimizations must be developed in "active" Stream APIs instead of "sufficiently-smart compilers".

7 Credits

The contents of this doctoral dissertation are based on published papers that were written in collaboration with others. Specifically:

– *Clash of the Lambdas* [5]; joint research with Nick Palladinos and Yannis Smaragdakis.
– *Streams à la carte* [4]; joint research with Nick Palladinos, George Fourtounis and Yannis Smaragdakis.
– *Recaf: Java Dialects As Libraries* [3]; work done while the author was affiliated with CWI; original design and implementation by the author, Pablo Inostroza and Tijs van der Storm; implementation of expression-level extensibility and corresponding applications by Pablo Inostroza.
– *Stream Fusion, to Completeness* [10]; original design by Oleg Kiselyov with help by the author on implementation and evaluation, jointly with Nick Palladinos and Yannis Smaragdakis.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs (1985)
2. Backus, J.: Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. Commun. ACM 21(8), 613–641 (Aug 1978)

3. Biboudis, A., Inostroza, P., Storm, T.v.d.: Recaf: Java dialects as libraries. In: Proc. of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 2–13. GPCE '16, ACM (2016)
4. Biboudis, A., Palladinos, N., Fourtounis, G., Smaragdakis, Y.: Streams à la carte: Extensible Pipelines with Object Algebras. In: Proc. of the 29th European Conference on Object-Oriented Programming. pp. 591–613. ECOOP '15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2015)
5. Biboudis, A., Palladinos, N., Smaragdakis, Y.: Clash of the lambdas. In: Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems. ICOOOLPS '14 (2014)
6. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming. pp. 315–326. ICFP '07, ACM (2007)
7. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture. pp. 223–232. FPCA '93, ACM (1993)
8. Kelsey, R., Hudak, P.: Realistic compilation by program transformation (detailed summary). In: Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 281–292. POPL '89, ACM (1989)
9. Kiselyov, O.: The Design and Implementation of BER MetaOCaml. In: Proc. of the 12th International Symposium on Functional and Logic Programming. pp. 86–102. FLOPS '14, Springer (2014)
10. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 285–299. POPL '17, ACM (2017)
11. Murray, D.G., Isard, M., Yu, Y.: Steno: Automatic Optimization of Declarative Queries. In: Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–131. PLDI '11, ACM (2011)
12. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: Proc. of the 26th European Conference on Object-Oriented Programming, ECOOP '12, vol. 7313, pp. 2–27. Springer Berlin Heidelberg (2012)
13. Oliveira, B.C.d.S., van der Storm, T., Loh, A., Cook, W.R.: Feature–Oriented Programming with Object Algebras. In: Proc. of the 27th European Conference on Object-Oriented Programming. pp. 27–51. ECOOP '13, Springer-Verlag (2013)
14. Paleczny, M., Vick, C., Click, C.: The java hotspot^{TM} server compiler. In: Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1. pp. 1–1. JVM'01, USENIX Association (2001)
15. Palladinos, N., Rontogiannis, K.: Linqoptimizer. `https://github.com/nessos/LinqOptimizer` (2013)
16. Peyton Jones, S., Tolmach, A., Hoare, T.: Playing by the rules: Rewriting as a practical optimisation technique in GHC. In: Haskell workshop. vol. 1, pp. 203–233 (2001)
17. Prokopec, A., Petrashko, D.: ScalaBlitz: Lightning-Fast Scala collections framework. `http://scala-blitz.github.io/` (2013)
18. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In: Proc. of the 9th International Conference on Generative Programming and Component Engineering. pp. 127–136. GPCE '10, ACM (2010)
19. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming. pp. 124–132. ICFP '02, ACM (2002)

20. Svensson, B.J., Svenningsson, J.: Defunctionalizing Push Arrays. In: Proc. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 43–52. FHPC '14, ACM (2014)

Coupled semiconductor lasers and their applications in telecommunications and networks

Michail Bourmpos*

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
mmpour@di.uoa.gr

Abstract. The aim of this thesis is to theoretically and experimentally investigate the nonlinear dynamics of coupled semiconductor lasers, in various network topologies and under different operating conditions. The nodes of the aforementioned networks proved capable of exhibiting synchronized chaotic optical outputs and in special cases at zero-lag. The first large scale network implementation of bidirectionally coupled semiconductor lasers with long interacting cavities, is presented.

1. Introduction

Coupled oscillators are capable of producing diverse dynamics and therefore have been a topic of great interest, with many applications in fields such as cryptography [1], telecommunications [2], control engineering [3] and more. The collective behavior of coupled oscillators, in various network topologies, has also been investigated over the past decades, driven mainly by the fact that these networks can be directly associated with complex physical [4] or biological systems [5]. Semiconductor lasers (SLs) are known nonlinear elements that generate complex dynamics and have been extensively used as models in the aforementioned networks [6]. In the simplest case of a mutual interacting network, two mutually coupled identical SLs can produce generalized synchronized dynamics [7]. When a third - relay element is added between them, isochronous synchronization can be achieved [8]. Zero-lag synchronization has also been observed for larger networks with multiple nodes [6]. In the present work, we have investigated arithmetically and experimentally multi-nodal all-optical networks in various topologies based on mutual coupling, in terms of synchronization, complexity and robustness. The nodes are represented by typical semiconductor lasers.

The rate equation mathematical model has been used to describe the operation and dynamics of the nodes, which can be applied to any of the investigated network topologies. This model is formulated in vector form and is based on the Lang Kobayashi model [9], originating from the representation used in [6] and including frequency detuning terms among oscillators as in [10].

$$\frac{d\vec{E}(t)}{dt} = j\overrightarrow{\Delta\omega} \circ \vec{E}(t) + \frac{1}{2}(1 + j\alpha)\left(\vec{G}(t) - \frac{1}{t_{ph}}\right) \circ \vec{E}(t) \\ + K^{T}\left(\vec{E}(t-\tau)) \circ e^{-j\omega_0\tau}\right) + \sqrt{D}\,\vec{\xi}(t) \tag{1}$$

$$\frac{d\vec{N}(t)}{dt} = \frac{\vec{I}}{e} - \frac{\vec{N}(t)}{t_s} - \vec{G}(t) \circ \left|\vec{E}(t)\right|^2 \tag{2}$$

$$\vec{G}(t) = g_n\left(\vec{N}(t) - N_0\right) \circ \left(1 + s\left|\vec{E}(t)\right|^2\right)^{\circ -1} \tag{3}$$

The vectors of the SLs optical fields and carrier densities are $\vec{E}(t)=[E_1(t);E_2(t);\ ...\ ;E_n(t)]$ and $\vec{N}(t)=[N_1(t);N_2(t);\ ...\ ;N_n(t)]$ respectively. The vector of uncorrelated complex Gaussian white noises is represented by $\vec{\xi}(t)$. The time delays and

* Dissertation Advisor: Professor Dimitris Syvridis

couplings between the nodes of the network are kept in the *n* x *n* arrays *K* and *τ*, where the actual values from node *i* to node *j* are represented as $τ_{ij}$ and k_{ij} respectively. By appropriate manipulation of the coupling and time delay arrays (*K* and *τ)* we can construct the desired network topologies, adopting coupling asymmetries wherever needed. The result of the Hadamard product ($\circ$) between the *n* x *n* arrays of the time delayed optical field *E(t-τ)* and the phase shift $e^{-j\omega_0\tau}$ in equation (1), is also a *n* x *n* array, where the element *(i,j)* is the delayed optical field of *i* injected into *j,* followed by the corresponding phase shift, thus equal to $E_i(t-\tau_{ij})e^{-j\omega_0\tau_{ij}}$.Vector $\vec{I}$ of equation (2) contains the biasing current for all lasers which is set to *I=18mA* throughout this work, while the solitary lasing emission threshold is I_{th}*=17.4mA*. Each laser is detuned with respect to the reference laser frequency ω_0, at variable values $\Delta\omega_j$, included in the vector $\overrightarrow{\Delta\omega}$ of equation (1). The *n* x *1* vectors *α, t_{ph}* and *s,* include the linewidth enhancement factors, photon lifetimes and saturation gain coefficients of the *n* SLs respectively. Finally, vector $\left|\vec{E}(t)\right|^2 = [|E_1(t)|^2|E_2(t)|^2 \dots |E_n(t)|^2\,]$ contains the power of optical fields. The Hadamard inverse $(1+s\circ\left|\vec{E}(t)\right|^2)^{\circ-1}$ yields $1/(1+s_i|E_i(t)|^2)$ for the *ith* laser. SLs share the same values for the rest of their intrinsic parameters, so there parameters g_n and N_0 are not expressed in vector form, although this could also be possible.

Simulations were performed for the set of differential equations (1-3) using the 4th order Runge-Kutta method, with a time-step of 0.8psec. Optical power has been deducted from the complex optical field using the appropriate conversion [11].

In all network topologies presented below, we formulate the coupling matrix *K* and assume that the time delay matrix is similarly constructed, with constant values of *5ns* delays between the nodes. The frequency detuning from the reference laser frequency are randomly chosen, following a Gaussian distribution in the range of 2π·(±1GHz).

2. Star Network

For a star network of 50 remote nodes coupled through a central typical SL we formulate the 51x51 coupling matrix *K* as follows:

$$K = \begin{bmatrix} 0 & 0 & \dots & 0 & k_{1,51} \\ 0 & 0 & \dots & 0 & k_{2,51} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & k_{50,51} \\ k_{51,1} & k_{51,2} & \dots & k_{51,50} & 0 \end{bmatrix} \qquad (4)$$

The 51st row of the matrix is matched with the central node of the star topology. We have assumed identical couplings from the hub node to the star nodes ($k_{51,i} = k_{51,j} = k$) and from star nodes to the hub $k_{i,51} = k_{j,51} = \beta\cdot k$, where *0<β<1* is a coupling asymmetry coefficient we have introduced to keep the accumulated optical injection into the hub laser within reasonable range.

We must point out that for this topology the hub laser frequency detuning is assumed to be zero, without loss of generality.

For different values of the parameter pair *(k,β)*, a mapping of the mean zero-lag cross-correlation between all SL pairs *(i,j)* is constructed, as shown in figure 1.

Based on this mapping, we have selected the pair of parameters *{k=60ns^{-1} , β=0.5}* where high zero-lag mean cross-correlation is achieved ($C_{i,j}^{mean}$*=0.921*) in combination with higher complexity. For this pair of values, a sensitivity analysis, when a SL is added or subtracted from the network, is performed.

First we connect a new node to the network, with various coupling and time-delay parameter values. We are interested in whether the connection of this non-identical laser with unmatched operational parameters will influence the behaviour of the backbone network. Moreover, we would like to know the tolerance in the parameter mismatch in order for the SL to be synchronized with the rest of the nodes in the network and if this node-addition can somehow be detected.

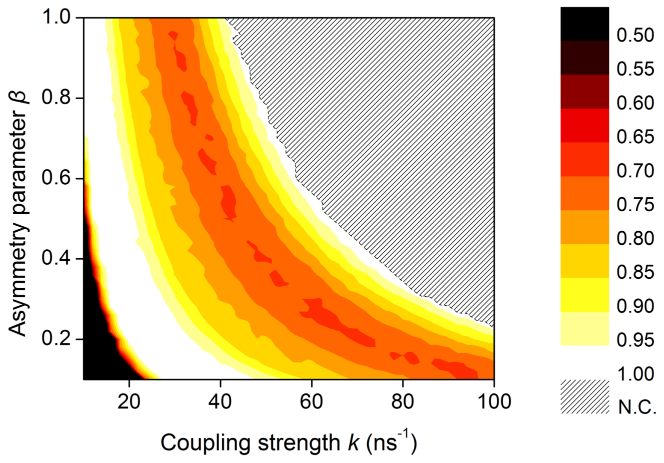


Figure 1. Mean zero-lag cross-correlation among the 50 star lasers.

For various values in the frequency detuning of the newly connected node and its time delay from the hub laser we calculate the change in the mean and minimum (worst case) zero-lag cross correlation of the original 50-node star network (figure 2).

It is evident that the network has absorbed the mismatch of the one additional laser, even if its detuning is significantly larger than 1GHz or its time-delay is different than *5ns*. The difference in the mean and minimum zero-lag cross-correlation is statistical and calculated to be$|\Delta C_{i,j}^{mean}|$*~0.013* and $|\Delta C_{i,j}^{min}|$*~0.021* respectively. The only parameter values of the added node for which the change is not statistical lie in the region of *τ=5ns* and for small frequency detuning values. The consistence of these values with the rest of the network parameters imposes a measurable positive change in the mean correlation value, making the synchronized new node detectable by the network.

Disconnecting a SL from the original network of 50 nodes is a rather more straightforward case. The mean correlation of the network after disconnecting the new laser is shifted now to a lower value by $\Delta C_{i,j}^{mean}$ = *-0.012*, attributed to a reduction of the coupling strength among the 49 lasers left within the network. However, simulation results prove that there is no dependence on the value of the disconnected laser's frequency detuning; the effect of disconnecting a node with large detuning seems almost equivalent to disconnecting a node with close to zero detuning.

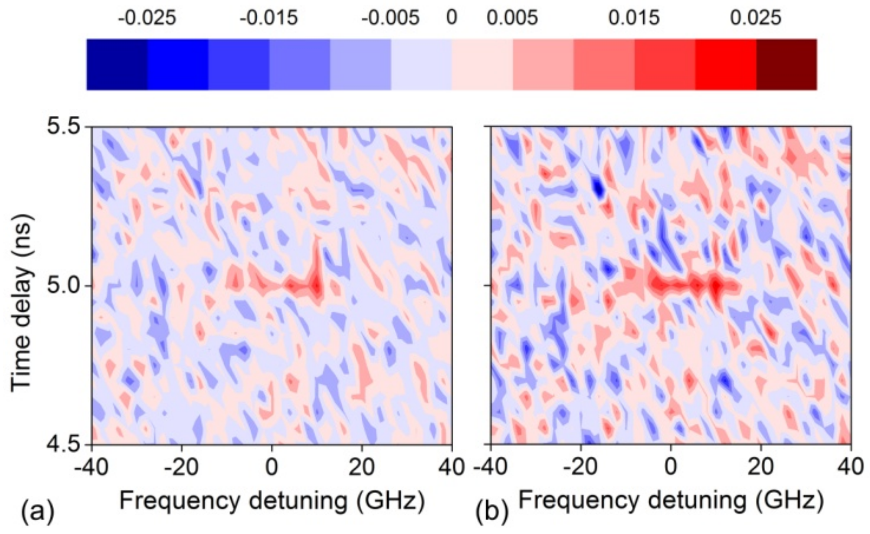


Figure 2. Change in the mean (a) and minimum (b) zero-lag cross correlation of the 50 star laser network, when an additional laser is connected to the network, for various values of frequency detuning and time delay of the added laser.

3. Mesh Network

For a fully-connected mesh network of 50 SLs, each one coupled to every other, we formulate the 50x50 coupling matrix *K* as follows:

$$K = \begin{bmatrix} k_{1,1} & k_{1,2} & \dots & k_{1,49} & k_{1,50} \\ k_{2,1} & k_{2,2} & \dots & k_{2,49} & k_{2,50} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ k_{49,1} & k_{49,2} & \dots & k_{49,49} & k_{49,50} \\ k_{50,1} & k_{50,2} & \dots & k_{50,49} & k_{50,50} \end{bmatrix} \qquad (5)$$

where $k_{i,i}=0$ for zero feedback of the SL nodes.

Assuming equal couplings $k_{i,j}=k, \forall i,j$ and for different values of *k* we plot the mean zero-lag cross-correlation between all SL pairs (figure 3). Based on this figure we choose $k=1.5ns^{-1}$ as the coupling between the nodes, which yields a mean zero-lag cross-correlation of $C_{i,j}^{mean}=0.964$ and then add a new node, with different parameter values, to the network.

Again, the network has absorbed the mismatch of the one additional laser (figure 4). The difference in the mean and minimum zero-lag cross-correlation is statistical and calculated to be $|\Delta C_{i,j}^{mean}|\sim 0.001$ and $|\Delta C_{i,j}^{min}|\sim 0.004$ respectively. The newly added SL produces a consistent, non-statistical increase in the mean and minimum zero-lag cross-correlation of the network, only when it's time-delay is equal to *5ns* - the common time-delay of all nodes - and for small values of frequency detuning. Similar results to those of the star network are obtained for the maximum value of the mean cross-correlation between the connected laser and the 50 SLs in the fully-connected mesh network, as well as for the corresponding time-lag, for various detuning and time delay values of the added node.

Disconnecting a node from the network leads to a detectable minor mean cross-correlation degradation, which is statistical and independent of the removed node's frequency detuning, as in the case of the star network.

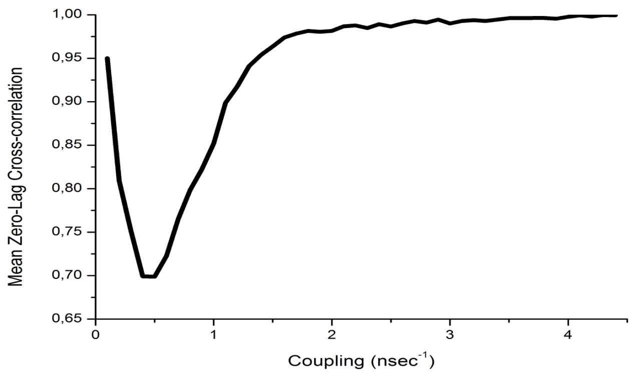


Figure 3. Mean zero-lag cross-correlation among the 50 star lasers.

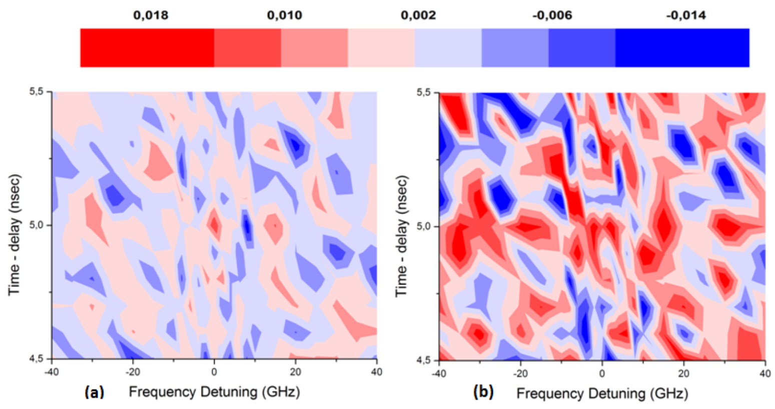


Figure 4. Change in the mean (a) and minimum (b) zero-lag cross correlation of the 50 laser fully-connected mesh network, when an additional laser is connected to the network, for various values of frequency detuning and time delay of the added laser.

4. Ring Network

For a ring network of 50 SLs, where every node is connected only to its two neighbors, the 50x50 coupling matrix *K* is formulated as follows:

$$K = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & \dots & 0 & 0 & k_{1,50} \\ k_{2,1} & k_{2,2} & k_{2,3} & \dots & 0 & 0 & 0 \\ 0 & k_{3,2} & k_{3,3} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & k_{48,48} & k_{48,49} & 0 \\ 0 & 0 & 0 & \dots & k_{49,48} & k_{49,49} & k_{49,50} \\ k_{50,1} & 0 & 0 & \dots & 0 & k_{50,49} & k_{50,50} \end{bmatrix} \tag{6}$$

with $k_{i,i}=0$ for zero feedback of the SL nodes, common couplings $k_{i,j}=k$, $\forall i, j$ and time delays of *5ns.* The nodes are sorted and positioned in the ring by increasing frequency detuning values.

For different coupling values we calculate the maximum cross-correlation between all node-pairs and the corresponding lag. Only the second nearest neighbors (*i-2* and *i+2* for the *i*th node) have consistently their maximum cross-correlation value at zero lag, even though this is also observed in several other 'even pairs' (*j* with *j+2·l*). We plot (figure 5) the maximum, mean and minimum of these cross-correlation (regardless of lag) values. Only for small couplings, where low complexity dynamics are observed, we obtain high cross-correlation values.

Ring network topologies exhibit poor synchronization quality and thus are of no particular interest regarding possible application in telecommunications or sensing.

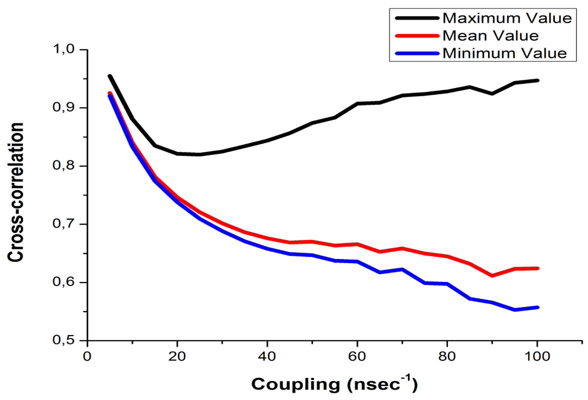


Figure 5. Maximum, mean and minimum cross correlation of all pairs in the 50 node fully-connected mesh network, regardless of lag.

5. Effect of parameters on synchronization

We have also investigated two different mutually coupled SL network topologies, with respect to the number of nodes and the discrepancies in key SL parameters. Star or fully-connected mesh networks with M=8, 16 and 24 nodes seem to be significantly affected by the differences in the SLs linewidth enhancement factor and photon life time parameters. The wider the range these different values are spread, the worse the efficiency of the synchronized dynamics. The same applies for discrepancies in the operating frequency of the SLs, as also shown in previous research. Differences in the saturation gain coefficient are unimportant as proven in this work. Furthermore, the full-mesh network topology seems to better cope with parameter mismatch, than the star network topology. Based on the above, we can propose that the full-mesh network topology should be adopted for a large-scale experiment of optical networks yielding synchronized complex dynamics. Given that the operating frequency can be adjusted by use of temperature control, it seems of the outmost importance to be able to manufacture and use SLs with similar intrinsic parameter values, especially for the linewidth enhancement factor and the photon life time, which have been proven to be of significance.

6. Bubbling

Differences in the internal characteristics of the SL nodes have a noteworthy impact on the performance and quality of network synchronization. For instance, networks with small differences in the SLs free-running frequencies require larger coupling strengths and/or smaller driving currents for adequate synchronization of the produced signals. In any case, de-synchronization windows of rather small duration - even at well-synchronized systems - have been observed. These intermittent desynchronizations are noise and/or parameter mismatch induced events, referred in literature also as bubbling [12,13]. They have been observed both as fast and frequent events in the coherence-collapse regime and as slower and less-frequent events in the low-frequency fluctuation (LFF) regime, in the case of bi-directionally coupled SLs [12,13].

In this work we have numerically investigated the behavior of bubbling effects in a well-synchronized SL network, adopting a star topology, and how these events determine the network's overall operation. We have concentrated on cases where chaotic dynamics are generated under strong node coupling. Zero-lag cross-correlation and synchronization error between pairs of star nodes in this network are estimated, for various conditions of delay between nodes, coupling strength and driving current of the mediating element (hub). In the context of these investigations, the statistical properties of the de-synchronization events are monitored and quantified, such as their reproducibility and duration. The general trends of bubbling statistics are finally associated with the physical phenomena caused by the changes in the network's critical variables.

The behavior of a star SL network has been examined, while altering certain key network parameters in terms of the de-synchronization events that appear in well-synchronized chaotic dynamics. Longer links between the star and hub nodes lead to longer and more infrequent de-synchronization events, keeping the overall synchronization at almost the same levels. On the contrary, the increase in coupling strengths leads to a more efficient synchronization with shorter and more infrequent bubbling events. Finally, the increase in the hub SLs driving current eventually leads to minimization or even elimination of the de-synchronization events by upgrading the role of the hub in the network operation, and interchanging its lagging with leading dynamics.

7. Experimental investigation

In the present work we have extended the generalized synchrony investigations in a network of up to 16 mutually-coupled identical SLs, connected through similar - yet unmatched - distant optical paths. We show that each unit's properties and operating parameters establish it as a member of the overall synchronized network, a member of intra-network synchronized clusters or just an outlier unit. Strict frequency matching (<200MHz) of the optical emitted signals allow synchrony at configurations with even a few identical SLs. In contrast, when non-identical SLs couple with the network they fail to synchronize at any operational condition. Moreover, when shifting identical SLs from a common emission wavelength (global operation) to multiplexed wavelengths (cluster operation), it is shown that the network can maintain intra-cluster synchrony. The latter property is validated for ultra-dense wavelength multiplexing of the coupled units, with chaotic carrier spectral distance of only 50pm.

The coupling topology follows the fully-connected SL architecture shown in figure 6a. Each laser is selected from a pool of identical SLs and emits to the network, while receiving from all counterparts - including its own signal - through a common tunable reflector. For up to 16 lasers (or else referred as network nodes) and no long-haul transmission path, one amplification stage provides sufficient power to the injected signals for laser synchrony. However, in the presented investigation two amplification stages are used so that a larger range of coupling strengths can be tested among the laser nodes. Optical filtering with 0.36nm (~40GHz) 3dB-bandwidth is used to reduce erbium-doped fiber amplifiers’ (EDFA) spontaneous emission, without imposing frequency-selective feedback conditions. Inline fiber power monitors (PM) display the circulating average optical power. The total round trip time of the cavities formed between pairs of lasers is 117.92 ± 0.12m. Each laser's optical output is monitored through isolated ports that eliminate any residual feedback. These outputs are used to screen the optical and microwave properties of the emitted signals through appropriate monitoring instrumentation.

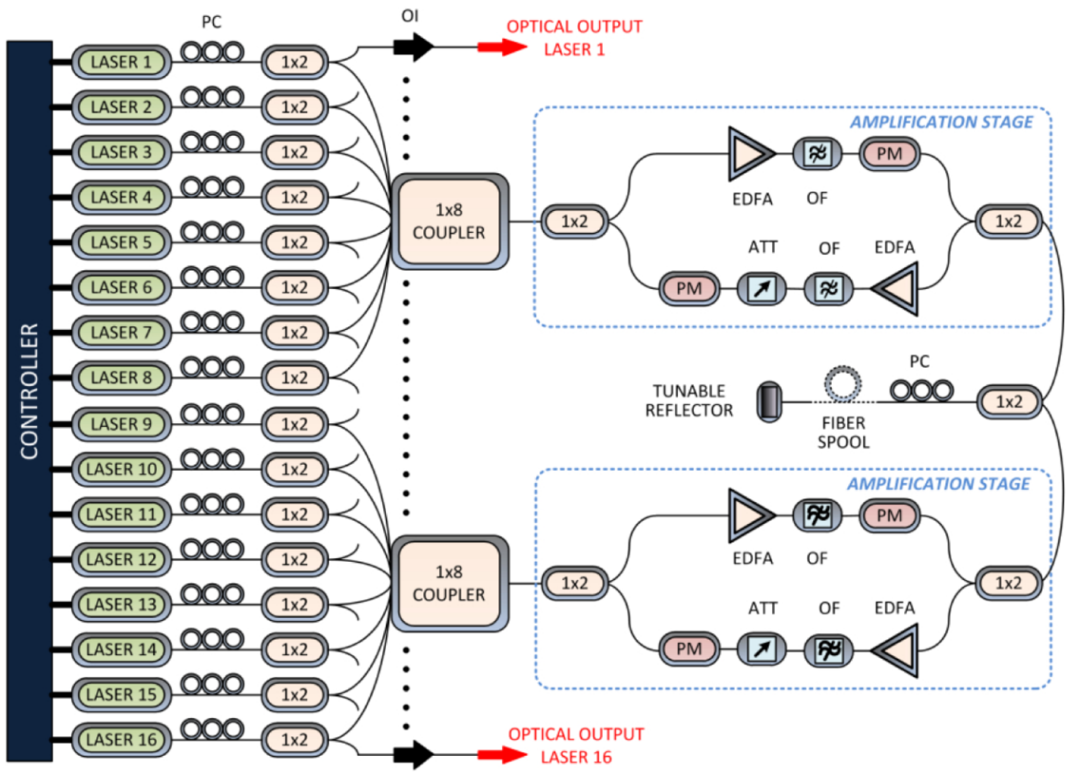


Figure 6. Full-mesh-type network with optically-coupled 16 SLs. Laser network topology: PC: Polarization controller, 1x2 and 1x8: optical couplers, EDFA: 25dB-gain Erbium-doped fiber amplifier, OF: Optical filter, PM: Inline optical power monitor, ATT: Optical attenuator.

The coupling strength among the SLs determines not only the synchrony performance but also shapes the emitted laser dynamics through which synchrony is achieved. The injection ratio $R^{L\#}$ is a measure of the coupling strength among the SLs and expresses the ratio of the optical power inserted into a laser divided by the optical power emitted by the same laser. Thus, for a given laser $L_\#$ that participates in the coupled network, it is defined as:

$$R^{L^\#} = C^2 \cdot \frac{P_{inj}^{tot}}{P_{em}^{L^\#}} \tag{7}$$

where C is the laser-fiber coupling loss, $P_{inj}{}^{tot}$ is the total optical power reaching laser $L_\#$ through the associated fiber path and $P_{em}{}^{L\#}$ is the optical emitted power of laser $L_\#$ measured at its output fiber tip. Coupling loss C between laser facet and fiber is a parameter which cannot be verified directly for each device, since all devices are fiber pigtailed. For the injection ratio estimation, a value of $C = 0.5$ is used for all lasers, according to the specifications given by the laser manufacturer.

As presented in figure 7, optical coupling affects signal emission from very low R values (as low as 0.001), forcing the 16 lasers to deviate from the continuous wave emission and oscillate in various dynamical states. Only when R>0.05 correlated chaotic emission for the overall network (*average-CC*>0.8) is observed among all coupled lasers (gray-marked region). Even slight mismatches in SLs' internal parameters, operational characteristics, optical emission frequencies, as well as small deviations from polarization alignment, may result in different levels of synchrony. In the example of figure 7, the laser pair $L_{\#1}$ - $L_{\#2}$ shows an *average-CC* above 0.93, while the laser pair $L_{\#6}$-$L_{\#7}$ shows an *average-CC* close to 0.86. The variance of each *average-CC* value is explained by the transversal instabilities of the synchronization manifold imposed by the overall network operation. The appearance of de-synchronization events, albeit always present, shows a dependence on the coupling strength among the laser nodes. Their duration and occurrence frequency shape the correlation and variance level for each coupling strength condition. In figure 8 such de-synchronization events are shown between two SLs ($L_{\#1}$ and $L_{\#2}$) emissions. Usually when power dropouts arise in the emitted dynamics, de-synchronization for a small period of time - of the order of ns - is present. The fact that this duration is significantly shorter than the period for which the two lasers preserve high-level of synchronization deems the change in the statistical metric of *averaged-CC* insignificant. This behavior refers to an optimally coupled and operated 16-SL network. If SLs are biased to favor LFF emission or unmatched operational conditions apply, the de-synchronization events last longer, affecting the overall synchronization level. Finally, for very strong injection ratios (above 1), increased instabilities are observed, accompanied by longer de-synchronization events or lower complexity attractors, periodic oscillations and even continuous wave operation.

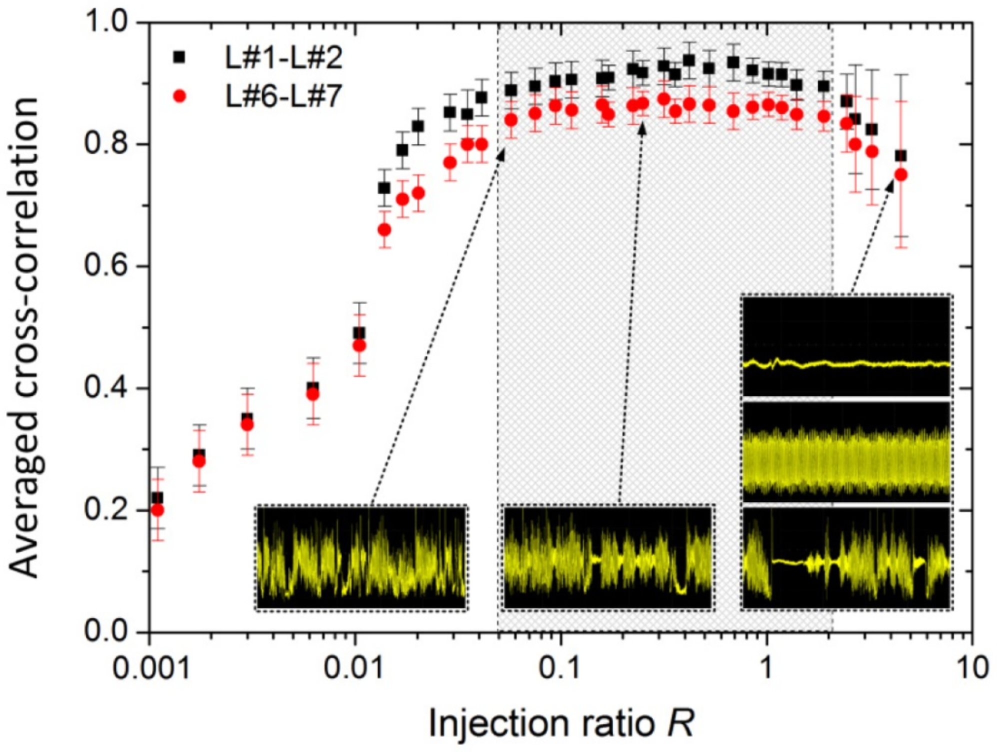


Figure 7. Effect of coupling strength on the correlated emission of a 16-laser coupled network. *Averaged-CC* values between two pairs of SLs (*L#1-L#2* and *L#6-L#7*) vs. the applied injection ratio, when the emitted power from the lasers is set to −15dBm. Timetraces in insets show the dynamics of the emitted signals for different coupling strengths. Gray region indicates synchronized network coupling conditions through chaotic signals

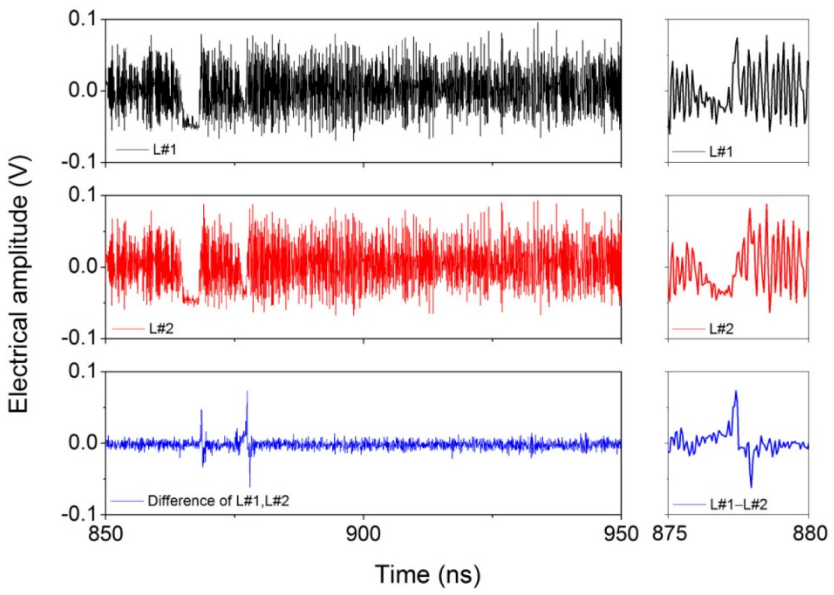


Figure 8. Temporal evolution of *L#1* and *L#2* emission, as well as their difference in a coupled 16-laser network, when *R* = 0.2dB. De-synchronization events appear when power dropouts occur and are minimized for optimal operating and coupling conditions. In the right column, a detail in the temporal region where a de-synchronization event takes place is provided.

8. Applications

Security

The concept of security in a 8-SLs network is tested by substituting one SL with a device provided from another manufacturer. In this investigation we study the potential of a user optically coupling with the network with a non-identical SL device to synchronize.

Wavelength emission is matched among all SLs, while the biasing current of the different SL and the optical injection level are varied so as to achieve the best synchrony level within the network. When considering moderate optical coupling (*R* = −9dB), the highest achieved *averaged-CC* between the non-identical SL and any of the identical SLs group is 0.34 at its most, as shown in figure 9a. It is obtained for the non-identical SL's near-threshold operation and is greatly lower than the worst synchronized pair within the identical SLs group (*averaged-CC*~0.82).

By enhancing the coupling conditions to *R* = −0.5dB, as presented in figure 9b, an equivalent behavior is observed. The only difference observed is the improved values of *averaged-CC* for the different (~0.62) and same manufacturers' (~0.89) SLs. Consequently, dissimilar hardware SL units hold by unauthenticated users fail to synchronize with the network at any operational condition. On the other hand, users with identical devices can access synchronized emission, as long as they select matched operating conditions and same dynamical regimes. Equivalent findings are also validated for a network containing two out of eight SLs from a different manufacturer.

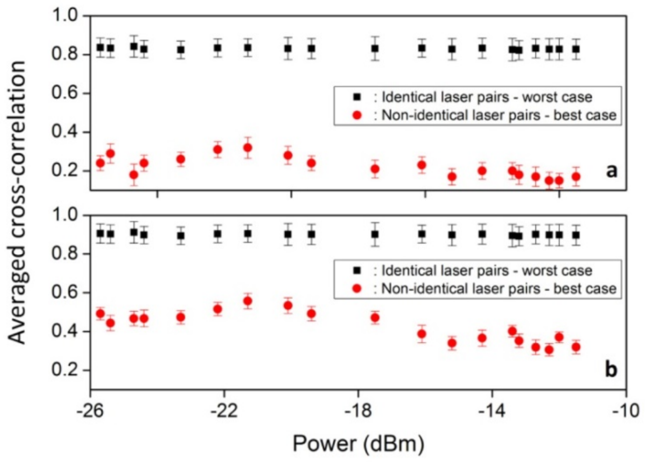


Figure 9. Synchrony comparison in an 8-node coupled network that includes 7 identical SLs and 1 SL from a different manufacturer. The comparison is made between the worst performance among the 7 identical lasers (black rectangles) and the best performance of synchronization between the different SL and the 7 identical lasers (red circles), versus the different SL's emitted optical power and for (a) $R = -9\text{dBm}$ and (b) $R = -0.5\text{dBm}$. All uncoupled identical lasers emit optical power −15dBm, while all uncoupled lasers operate at λ = 1549.600nm

Clustering

In a smaller network, we optically couple 8 SLs and examine the potential of the network SL nodes to obtain cluster synchrony by imposing frequency emission grouping. Initially, for small frequency detuning (<200MHz) of all 8 SLs - as in the 16-laser network case - optimized conditions lead to highly correlated emission. Specifically, pairwise *averaged-CC* values of at least 0.88, and as high as 0.97, are recorded for full-bandwidth detected signals, as shown in figure 10a. It becomes clear that by controlling frequency-matching conditions we can drastically reduce the least number of coupled nodes required for synchronized operation. By thermally shifting the emission wavelengths of the second quartet of lasers ($L_{\#5}$-$L_{\#8}$) by 50pm we obtain the correlation mapping if figure 10b. Synchrony is observed between lasers that participate in each cluster ($L_{\#1}$-$L_{\#4}$ and $L_{\#5}$-$L_{\#8}$) while correlated emission is always at very low levels when comparing inter-cluster nodes (*averaged-CC*<0.61). Thus, the same configuration can also lead to cluster synchronization.

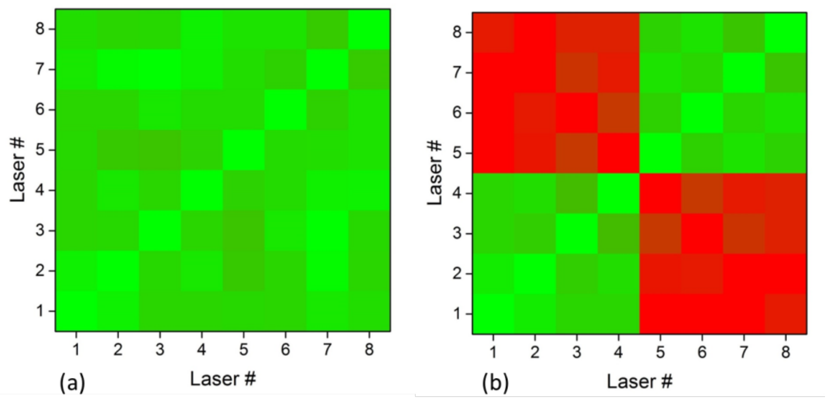


Figure 10. Cluster synchronization in an 8-SL coupled network configuration. Cross-correlation mapping with (a) zero-detuned wavelength laser emission, and (b) with cluster synchronization among two quartets of lasers ($L_{\#1}$-$L_{\#4}$ and $L_{\#5}$-$L_{\#8}$) that are 50pm spaced in wavelength.

9. Conclusions

In the present work, we have arithmetically and experimentally investigated multi-nodal all-optical networks, in various topologies and in terms of synchronization, complexity and robustness. We have presented the first large-scale implementation of 16 optically coupled and independently controlled SLs in a fully-connected synchronized network topology. The overall consistency of the synchronized network is profound albeit the presence of mismatched or disparate lasers interacting through optical coupling. Local instabilities causing short de-synchronization events do not annihilate the overall high level of synchrony.

Our work can be the basis on which advanced sensing and authentication protocols in future fiber-optic networks can be proposed. In an envisaged application based on the concept of this work, other real-life large-scale networks of coupled oscillators can be simulated through the use of SLs, exploiting the speed of phenomena evolution in such configurations, for prediction purposes.

REFERENCES

[1] Uchida, A.,, Amano, K., Inoue, M., Hirano, K., Naito, S., Someya, H., Oowada, I., Kurashige, Y., Shiki, M., Yoshimori, S., Yoshimura, K., and Davis, P., “Fast physical random bit generation with chaotic semiconductor lasers,” Nat. Phot. 2, 728–732 (2008).

[2] Argyris, A., Syvridis, D., Larger, L., Annovazzi-Lodi, V., Colet, P., Fischer, I., García-Ojalvo, J., Mirasso, C.R., Pesquera, L., and Shore, K.A., "Chaos-based communications at high bit rates using commercial fiber-optic links," Nature 438 (7066), 343-346, (2005).

[3] Fradkov, A.L., Evans, R.J., and Andrievsky, B.R., "Control of chaos: methods and applications in mechanics," Phil. Trans. R. Soc. A 364, 2279-2307 (2006).

[4] Strogatz, S.H., Abrams, D.M., McRobie, A., Eckhardt, B., and Ott, E., "Crowd synchrony on the millennium bridge," Nature 438 (7064), 43-44 (2005)

[5] Strogatz, S.H., and Stewart, I., "Coupled oscillators and biological synchronization,", Scient. Amer. 269, 68–73 (1993).

[6] Zamora-Munt, J. , Masoller, C., Garcia-Ojalvo, J. and Roy R., "Crowd synchrony and quorum sensing in delay-coupled lasers," Phys. Rev. Lett. 105, 264101 (2010).

[7] Mulet, J., Mirasso, C.R., Heil, T., and Fischer I., "Synchronization scenario of two distant mutually coupled semiconductor lasers", J. Opt. B: Quantum Sem. Opt. 6, 97 (2004).

[8] Zhou, B.B., and Roy R., "Isochronal synchrony and bidirectionalcommunication with delay-coupled nonlinear oscillators", Phys. Rev. E 75, 026205 (2007).

[9] Lang, R., and Kobayashi, K., "External optical feedback effects on semiconductor injection laser properties," IEEE J. Quantum Electron. 16, 347-355 (1980).

[10] Fischer, I., Vicente, R., Buldú, J.M., Peil, M., Mirasso, C.R., Torrent, M.C., and García-Ojalvo, J., "Zero-Lag Long-Range Synchronization via Dynamical Relaying," Phys. Rev. Lett. 97, 123902 (2006).

[11] Petermann, K. [Laser Diode Modulation And Noise], New Ed. Kluwer Academic Publishers Group, Netherlands (1991).

[12] J. Tiana-Alsina, K. Hicke, X. Porte, M. C. Soriano, M. C. Torrent, J. Garcia-Ojalvo and I. Fischer, "Zero-lag synchronization and bubbling in delay-coupled lasers", *Phys. Rev. E* vol. 85, 026209, 2012.

[13] V. Flunkert, O. D’Huys, J. Danckaert, I. Fischer, and E. Schöll, "Bubbling in delay-coupled lasers", *Phys. Rev. E* vol. 79, 065201, 2009.

A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling

Elias Konstantinidis [⋆]

National and Kapodistrian University of Athens
Department of Informatics and Telecommunications
`ekondis@di.uoa.gr`

Abstract. Over the last decade GPUs have been established as compute accelerators. However, GPU performance is highly sensitive to many factors, e.g. memory access patterns, branch divergence, the degree of parallelism and potential latencies. Consequently, the execution time on GPUs is a difficult to predict measure. Unless the kernel is latency bound, a rough estimate of the execution time on a particular GPU could be provided by applying the roofline model. Though this approach is straightforward, it cannot not provide accurate prediction results. In this thesis, after validating the roofline principle on GPUs by employing a micro-benchmark, an analytical performance model is proposed. In particular, this improves on the roofline model following a quantitative approach and a completely automated GPU performance prediction technique is presented. In this respect, the proposed model utilizes micro-benchmarking and profiling in a "*black-box*" fashion as no inspection of source/binary code is required. It combines GPU and kernel parameters in order to characterize the performance limiting factor and to predict the execution time, by taking into account the efficiency of beneficial computational instructions. In addition, the "*quadrant-split*' visual representation is proposed, which captures the characteristics of multiple processors in relation to a particular kernel. The experimental evaluation combines test executions on stencil computations, matrix multiplication and a total of 28 kernels of the Rodinia benchmark suite. The observed absolute error in predictions was 27.66% in the average case. Special cases of mispredicted results were investigated and justified. Moreover, the aforementioned micro-benchmark was used as a subject for performance prediction and the exhibited results were very accurate. Furthermore, the performance model was also examined in a cross vendor configuration by applying the prediction method on the AMD HIP/ROCm programming environment. Prediction errors were comparable to CUDA experiments despite the significant architectural differences of different vendor GPUs.

Keywords: performance model, GPU, roofline model

[⋆] Dissertation Advisor: Yiannis Cotronis, Associate Professor

1 Dissertation Summary

1.1 Introduction

The main focus of GPU computing is about performance so it would be of great significance to be able to predict performance of GPU applications on a wide range of hardware. Performance modeling information is particularly important that can be exploited for either the consideration of a hardware upgrade or even on taking important optimization decisions. However, performance impact of migrating to a GPU accelerator or moving from one type of GPU to another can be a puzzling process to predict. Performance bottlenecks can be different due to architectural differences or variations on the balance of processor resources between different types of processors.

CPUs do not require a vast amount of parallelism in order to yield decent performance. They utilize large cache memory hierarchies that are able to alleviate the long access latencies of main memory. In addition, they employ advanced techniques in order to maximize the single threaded performance, e.g. aggressive speculative execution, register renaming, result value forwarding, etc. All these features potentially eliminate pipeline and memory bottlenecks, leading to more predictable execution results.

On the other hand, GPUs are significantly more performance sensitive to supplied parallelism, resource usage and memory access patterns. They are considered as massively parallel compute devices as they practically need thousands of active threads in order to keep them occupied. This fact poses large problems with abundant parallelism as a requirement. The GPUs feature much smaller cache memories which in conjunction with the large amount of active threads allows only limited use, mostly for exploiting the spatial locality between sibling threads. The miss of large cache hierarchies forces programmers to effectively use main memory. However, GPUs require regular memory accesses with specific requirements in order to apply coalescing, which is a mandatory requirement for efficient memory accessing. All reasons above induce potential bottlenecks for GPU performance. Practical experience has proven that GPU performance is sensitive to design decisions and fine tuning. In general, GPUs tend to be less tolerant to naive programming practices in regard to performance. Overall, though GPUs provide great compute performance, this can only be achieved on problems that match their characteristics.

For all the reasons above this thesis is focused on proposing a performance model that provides the necessary abstraction in order to be applicable on a wide range hardware, yet it provides decent prediction accuracy, is quick and straightforward to apply and can be fully automated based on *black-box* kernel inspection. In addition, this model was developed as roofline based and as such it is able to indicate an upper bound on performance, which can be fairly useful to the programmer as a guidance, providing performance feedback for further optimizations. The ultimate goal was to provide a tool that runs automatically the whole performance prediction process by utilizing an existing GPU program and producing the final results without user's intervention.

1.2 Related work

The roofline model [12] introduced by Williams and Patterson, is a visual model that provides insight on the maximum expected performance of a kernel by considering both pure computation and DRAM memory transfer requirements. It is based on the assumption that performance is either bound on the compute potential or the memory bandwidth of the underlying processor. The performance bound is either one depending on the relative requirements of operations of the application. Operational intensity is measured in *flop/byte* units and is used to determine the limiting performance factor on a particular processor. This can be applied by estimating the program's operational intensity which is determined by the program's requirements as formula (1) indicates:

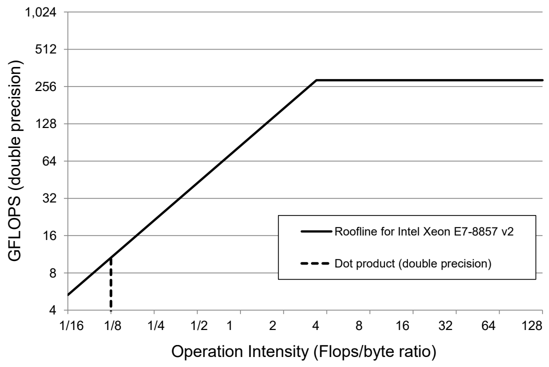


Fig. 1. The roofline visual model for Intel Xeon E7-8857 v2.

$$O_{kernel} = \frac{Operations_{(compute)}}{Traffic_{(memory)}} \tag{1}$$

The operational intensity is measured in *flop/byte* units and it is dependent on the application characteristics. Depending on the whether $O_{kernel} > \frac{Throughput_{dev}}{Bandwidth_{dev}}$ the kernel is considered as compute bound or memory bound. The graphical representation of the roofline model is able to provide a quick and insightful visual representation of the device theoretical peak performance. In figure 1 the solid line represents the theoretical peak performance of an Intel Intel Xeon E7-8857 v2 CPU depending on the program's operational intensity. In this example for program operational intensities up to 3.39 flop/byte the program is considered as memory bound. Compute bound programs must exhibit higher compute intensity.

1.3 Results

The proposed performance model, which is the primary contribution presented within this thesis, is an analytical GPU performance model based on the roofline model [12]. An early foundation of the proposed model was presented in a preliminary stage as a regular conference paper [7] and subsequently it was extended and published as an elaborate work in the form of a journal article [11]. The first paper contribution [7] presented an initial form of the method along with a limited number of experimental results. The relevant journal publication [11] extended the method to a fully automated prediction process. The experimental results included executions on a wide range of different real world kernels and a micro-benchmark. The hardware used for the experiments included 4 consumer and 2 professional GPUs. Furthermore, the proposed model was extended to the experimental use on a cross-vendor GPU environment by employing an AMD GPU and the exhibited results were quite promising.

Other contributions that have been used in this thesis include an implementation of a red-black SOR stencil computation method [9, 10] which has been utilized in the experiments and it poses as a proof of concept case study in this thesis. The reordering by color strategy was the primary contribution of this published work. A theoretical performance analysis of the algorithm was provided and the implementations included various kernels, each utilizing a different memory caching approach. Additionally, a set of developed LMSOR stencil computations [4–6] were also developed which served to investigate the re-computation strategy as an optimization. In this respect various implementations were investigated characterized by different operational intensities due to the different degree of re-computation applied. Implementations of this work were also applied on the performance model in this thesis. Last, a set of micro-benchmarks [8] was presented that serves to the purpose of better understanding of the hardware capabilities regarding the GPU's fast on-chip memories. The micro-benchmarks assess the fast on-chip memories which include shared memory, L1 & L2 cache, texture cache and constant memory cache.

2 Results and Discussion

2.1 The *quadrant-split* visual representation

The roofline visual model is a valuable abstract representation of the compute device capability. As an alternative representation, the *quadrant-split* is proposed where in the horizontal axis the memory bandwidth is used instead of the operational intensity. In this respect, a device can be represented by a single point on the chart determined by its memory bandwidth and compute throughput peak rates. A program can be represented by a half-line crossing the intersection of the axes with a slope equal to its operational intensity. The half-line is the visual bound for the distinction of the area into two parts where the kernel is expected to behave as memory bound for the devices residing in the upper half-quadrant and as compute bound for the others instead. For instance, figure 2 represents the LBMHD problem with respect to 4 GPUs and a CPU. The dashed arrow lines point to the estimated roofline performance points for the each device on the particular problem.

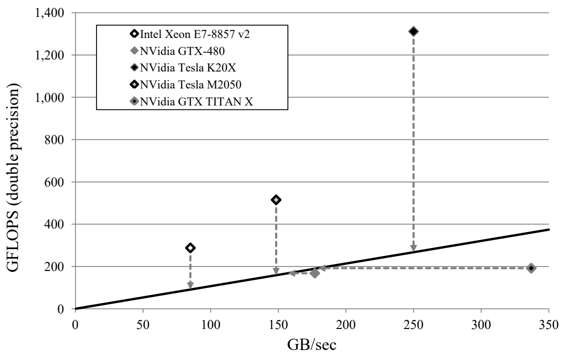
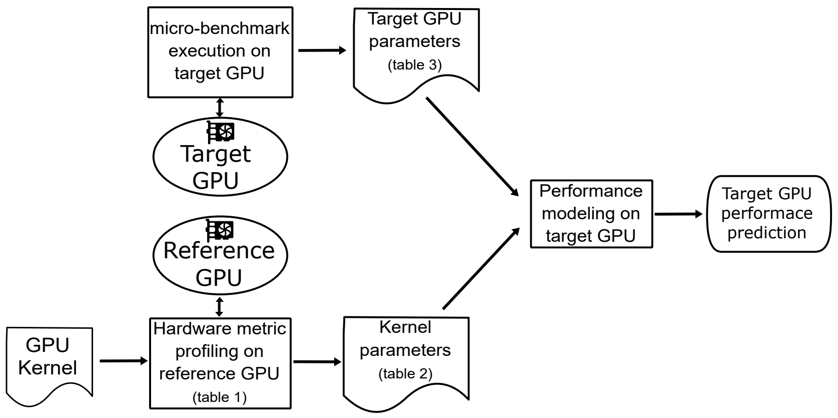


Fig. 2. The *quadrant-split* representation of the LBMHD problem using 5 CPU/GPUs.

2.2 The proposed performance prediction method

A profiling approach on a reference GPU is employed by extracting kernel execution information without requiring any internal knowledge of the kernel characteristics. The parameters used for the GPU device that is targeted for performance prediction are extracted by running a set of micro-benchmarks. The combination of both sets of parameters are employed for the performance prediction procedure. The whole process involves the steps described in figure 3.



micro-benchmark execution on target GPU
Target GPU parameters (table 3)
Target GPU
Reference GPU
GPU Kernel
Hardware metric profiling on reference GPU (table 1)
Kernel parameters (table 2)
Performance modeling on target GPU
Target GPU performace prediction

Fig. 3. The performance prediction methodology flow diagram.

In general, the approach for performance estimation of GPU kernels can be summarized in three aspects:

– Modeling compute and memory parameters of GPU kernels, largely independently of GPU architectural details, obtained by using a “*black box*” approach based exclusively on profiling measures (figure 3: "Hardware metric profiling on reference GPU")

- Modeling the GPU generic peak performance ratings on various operations, obtained by micro-benchmarking the target GPU (figure 3: "micro-benchmark execution on target GPU")
- Estimation of the target GPU performance (figure 3: "Performance modeling on target GPU") on the particular kernel according to:
  - the estimated maximum rate of executed compute operations on the target GPU for the particular kernel, and
  - the compute and memory demands of the given kernel (i.e. operational intensity) determining whether its performance is limited by the compute or memory throughput when executed on the target GPU

Kernel parameter extraction The required kernel parameters are extracted by profiling the execution of the subject kernel on a reference GPU. The list of the required kernel metrics is shown in table 1 and the provided notation will be used for reference.

Table 1. The NVidia profiler metrics required for the derivation of kernel parameters.

Metric	Notation	Description
flop_count_sp_fma	M_{fma32}	Number of single-precision floating-point multiply-accumulate operations executed by non-predicated threads
flop_count_dp_fma	M_{fma64}	Number of double-precision floating-point multiply-accumulate operations executed by non-predicated threads
inst_compute_ld_st	M_{ldst}	Number of compute load/store instructions executed by non-predicated threads
inst_executed	M_{inst}	The number of instructions executed
inst_fp_32	M_{fp32}	Number of single-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_fp_64	M_{fp64}	Number of double-precision floating-point instructions executed by non-predicated threads (arithmetic, compare, etc.)
inst_integer	M_{int}	Number of integer instructions executed by non-predicated threads
dram_read_transactions	$M_{tran\text{-}r}$	Device memory read transactions
dram_write_transactions	$M_{tran\text{-}w}$	Device memory write transactions

The produced parameter set is provided in table 2. K_{type} parameter determines the type of beneficial operations within the kernel. It can be either *fp64*, *fp32* or *int*. A simple rule based approach in order to avoid user interaction is a function selecting *fp64* if the M_{fp64} metric is non zero, *fp32* if the M_{fp32} is non zero or *int* otherwise. The W_{comp} parameter represents the total beneficial compute operations performed by the kernel. It is evaluated by formula (2).

Table 2. The set of required *kernel parameters* in the proposed performance model.

Parameter	Description	Obtained
K_{type}	Dominant ops (fp64, fp32 or int)	rule based function
W_{comp}	Compute operations	formula (2)
W_{traf}	DRAM bytes accessed	formula (3)
E_{mix}	Operation mix efficiency (%)	formula (4)
D_{ops}	Operation instruction density (%)	formula (5)
D_{ldst}	Ld/St instruction density (%)	formula (6)
D_{other}	Other instruction density (%)	formula (7)

$$W_{comp} = \begin{cases} M_{fp32} + M_{fma32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64} + M_{fma64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{2}$$

The parameter regarding the conducted memory traffic is the W_{traf} and it is estimated by using the DRAM transaction count metrics as shown in formula (3):

$$W_{traf} = 32 \times (M_{tran\text{-}r} + M_{tran\text{-}w}) \tag{3}$$

The efficiency of compute instructions E_{mix} is defined as shown in formula (4) which involves the type of compute instructions executed.

$$E_{mix} = \begin{cases} \frac{M_{fp32}+M_{fma32}}{2\times M_{fp32}} \times 100\%, & \text{if } K_{type} = \text{fp32} \\ \frac{M_{fp64}+M_{fma64}}{2\times M_{fp64}} \times 100\%, & \text{if } K_{type} = \text{fp64} \\ 50\%, & \text{if } K_{type} = \text{int} \end{cases} \tag{4}$$

Finally, the instructions executed are classified in 3 different types (compute, load/store and other instructions) and the individual density of each type in the instruction stream is determined by formulae (5), (6) and (7):

$$D_{ops} = \frac{I_{ops}}{I_{total}} \times 100\% \tag{5}$$

$$D_{ldst} = \frac{M_{ldst}}{I_{total}} \times 100\% \tag{6}$$

$$D_{other} = 100\% - D_{ops} - D_{ldst} \tag{7}$$

where I_{ops} and I_{total} are estimated by formulae (8) and (9):

$$I_{ops} = \begin{cases} M_{fp32}, & \text{if } K_{type} = \text{fp32} \\ M_{fp64}, & \text{if } K_{type} = \text{fp64} \\ M_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{8}$$

$$I_{total} = 32 \times M_{inst} \tag{9}$$

Target GPU parameter extraction All required device parameters are collected by using micro-benchmarks and are shown in table 3. All floating point computation throughput parameters (T_{SP} and T_{DP}) concern MAD (Multiply-ADd) operations. The T_{xxx} parameters (T_{SP}, T_{DP}, T_{int}, T_{add}, T_{ldst}) regard the compute throughput of the device in various types of instructions and the B_{mem} parameter which reflects the effective memory bandwidth of the device.

Table 3. The set of *GPU parameters* used in the performance model.

Parameter	Description	Unit
T_{SP}	Single precision floating point operation throughput	GFLOPS
T_{DP}	Double precision floating point operation throughput	GFLOPS
T_{int}	Integer multiply-add operation throughput	GIOPS
T_{add}	Integer addition operation throughput	GIOPS
T_{ldst}	Load/Store instruction throughput on shared memory	GOPS
B_{mem}	Memory bandwidth	GB/sec

Kernel performance estimation In this model the throughput of various instruction types is considered for the efficiency estimation of instruction execution regarding beneficial computation. The purpose is to estimate the attainable peak throughput by considering the portion in which the pipeline is available for the execution of beneficial instructions. In this regard the instruction type densities (D_{ops}, D_{ldst}, D_{other}) should be considered in order to provide an estimation on the overall instruction execution throughput on the particular kernel.

The peak throughput on raw beneficial operations is selected in (10):

$$T_{op} = \begin{cases} T_{SP}, & \text{if } K_{type} = \text{fp32} \\ T_{DP}, & \text{if } K_{type} = \text{fp64} \\ T_{int}, & \text{if } K_{type} = \text{int} \end{cases} \tag{10}$$

For the estimation of the instruction execution efficiency the instruction densities along with the instruction throughput for various types are considered. The instruction types considered correspond to the throughput parameters of the GPU (table 3). The fastest instruction on the GPU typically is the single precision multiply-add instruction, and therefore it is the instruction that potentially is used to execute the most operations per second. So, the single precision multiply-add instructions are used as a point reference. The weight factor of executing a type of instruction is defined as the throughput ratio of fast single precision floating point instructions to the throughput of the particular type of instructions. Thus, weight factor is normalized by setting the weight of single precision instructions to 1. Therefore, the weight of all other instructions is typically greater or equal to 1. In this regard we define the weight factor operators as follows in formulae (11), (12), (13):

$$W_{op} = \frac{T_{SP}}{T_{op}} \quad (11) \qquad W_{ldst} = \frac{1/2 T_{SP}}{T_{ldst}} \quad (12) \qquad W_{other} = \frac{1/2 T_{SP}}{T_{add}} \quad (13)$$

In the estimation of W_{other} the throughput of integer addition is used. This is an arbitrary decision based on the assumption that the rest of the instructions apart from computation and load/store, is constituted mostly of simple integer instructions or instructions that execute roughly with the same cost. The ½ factor in (12) and (13) is applied in order to convert the operation throughput rate T_{SP} to instruction execution rate as each floating point MAD instruction is accounted as 2 operations. All beneficial operations are assumed to be executed using MAD instructions (two operations per instruction) whereas the load/store and integer addition operations are assumed to be implemented with single operation instructions. By taking into account the instruction densities and the respective weight factors the relative execution cost of each instruction type can be defined as shown in (14), (15), (16):

$$C_{op} = D_{ops} \times W_{op} \tag{14}$$

$$C_{ldst} = D_{ldst} \times W_{ldst} \tag{15}$$

$$C_{other} = D_{other} \times W_{other} \tag{16}$$

The estimated instruction efficiency can be estimated by formula (17):

$$E_{instr} = \frac{C_{op}}{C_{op} + C_{ldst} + C_{other}} \times 100\% \tag{17}$$

This cost modeling for the instruction execution assumes that all instructions are executed by the GPU multiprocessor on a single pipeline and therefore the execution of different types of instructions cannot be co-issued in a super-scalar fashion.

The adjusted throughput is estimated by applying both efficiency ratios each decreasing the theoretical instruction throughput by a factor. The adjusted throughput is given in (18):

$$T'_{op} = E_{mix} \times E_{instr} \times T_{op} \tag{18}$$

As such, the kernel's operational intensity is $O_{krn} = W_{comp}/W_{traf}$ and the device's adjusted operational intensity is $O_{dev} = T'_{op}/B_{mem}$. The comparison of the two intensities is used to determine whether the application is considered to behave as memory or compute bound. Thus, the estimated compute throughput is given by (19):

$$T_{predicted} = \begin{cases} T'_{op}, & \text{if } O_{krn} > O_{dev} \\ O_{krn} \times B_{mem}, & \text{if } O_{krn} \leq O_{dev} \end{cases} \tag{19}$$

2.3 Experimental evaluation

The executed experiments include two variants of stencil computations (red/black SOR & LMSOR) [9, 10], a matrix multiplication (SGEMM) kernel and a large subset of the Rodinia benchmark suite[3]. The experiments were applied on 6 different GPUs, characterized by 4 different architectures. The prediction procedure on red/black stencil computations, SGEMM and Rodinia benchmarks exhibited an average APE 3.42%, 15.18% and 28.97%, respectively. By summarizing all prediction results it is concluded that out of all conducted experiments more than half of them exhibited less than 25% APE (Absolute Percentage Error). This is considered a significant achievement given the small set of input that is used by the method.

In order to assess the performance prediction method in a cross vendor environment, the HIP/ROCm platform of AMD was chosen because of its CUDA kernel source code compatibility feature. By porting a CUDA implementation to HIP, the kernel source code effectively remains the same. This allows the profiling procedure to be performed on NVidia hardware by either using the CUDA application or the HIP application itself as HIP provides a compatibility layer for both hardware platforms.

The HIP programming environment was applied as supported by ROCm 1.4.0 release, on Ubuntu 14.04 Linux 64bit, using an AMD R9-Nano GPU. The kernel parameters were extracted on the GTX-480 and used for the performance prediction model on the AMD GPU. The required benchmarks were also ported to HIP platform and they were used to generate the R9-Nano GPU parameters.

The applied problems were the red/black SOR stencil computation, SGEMM and the lavaMD benchmark from the Rodinia suite (*lvmd-krn*). Running the performance model yields the execution times shown in table 4. It is evident that the observed prediction errors are very comparable to the ones produced on NVidia GPUs. Out of the 3 kernels, *lvmd-krn* exhibited slightly higher APE.

Table 4. Prediction results on the R9-Nano GPU for red/black SOR, SGEMM and lvmd-krn kernels

Benchmark	Predicted time (msecs)	Measured time (msecs)	Error (%)
red/black SOR	7.75	8.72	-11.18%
SGEMM	0.83	0.94	-11.45%
lvmd-krn	46.27	54.57	-15.21%

In general, it is expected to observe slightly higher APEs on the AMD platform due to the architectural differences between the two different vendor GPU architectures. These differences could slightly differentiate the extracted kernel parameters between the two architectures. However, this is not expected they change dramatically allowing the use of the performance prediction method in cross architecture environments, in the same way it was applied on this experiment.

3 Conclusions

This thesis presents an analytical performance model that derives from the roofline model [12]. Through a quantitative approach, the proposed model is able to provide timings that approximate actual execution measurements on real hardware. In addition, an alternative visual representation approach was presented, named *quadrant-split*, which is insightful in cases of multiple compute devices being represented along with a single application characterized by a particular operation intensity. The merit of the model's simplicity and its high abstraction characteristic allows providing results, final and intermediate, that can be easily interpreted by the final developer by being more human friendly. The small amount of required parameters pose the method as readily applicable.

One of the key points of the proposed method is the ability to extract the kernel's parameters by exploiting a mere set of profiling metrics as input parameters. This is captured through a kernel profiling procedure in a *black-box* fashion. Any internal knowledge of the kernel structure itself is not required by the developer. Furthermore, the proposed method can be developed as an automated tool which is executed without intervention from the developer. In this regard, the developer can apply the method on kernels and use it as a guidance tool without previous inspection the kernel design itself.

The proposed method achieves a better understanding of both compute and memory workloads compared to a pure theoretical peak approach primarily for two reasons. First, both the execution of non-essential and load/store instructions are considered by modeling their implications in the instruction pipeline and thus, their impact on the effective peak performance on beneficial instructions. Additionally, the type of mix of compute operations is also taken into account, i.e. the proportion of effective multiply-add operations in total amount of compute instructions. Second, the memory traffic requirements are measured by considering the actual traffic, thus any trivial locality and the degree of memory access coalescing are being indirectly accounted. The model provides an adjusted roofline on the peak performance based on these considerations.

The proposed performance model was tested and validated on a wide range of real world kernels. It was applied on stencil computations (red/black SOR and LMSOR), matrix multiplication and a wide range of Rodinia suite kernels. Furthermore, it was also tested for cross-vendor applicability on the HIP programming environment [2, 1] of the ROCm platform which is developed by AMD. The results were quite promising as they were similar to CUDA prediction in terms of absolute errors, despite the broader architectural differences between different vendor GPUs. The exact performance is dependent on special issues as the exact instruction mix, variability in cache behavior, pipeline latencies, the available parallelism and additional latencies which push performance to lower levels than the predicted ones, as the proposed model does not take into account these factors. Nevertheless, in these cases the performance prediction measurements serve as an upper bound performance and they indicate the potential room for improvement with further optimizations.

References

1. AMD. HIP. `https://github.com/GPUOpen-ProfessionalCompute-Tools/HIP`, 2016.
2. AMD. *HIP Data Sheet*, 2016. Rev. 1.7.
3. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.
4. Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. *Parallel SOR for Solving the Convection Diffusion Equation Using GPUs with CUDA*, pages 575–586. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
5. Yiannis Cotronis, Elias Konstantinidis, Maria A. Louka, and Nikolaos M. Missirlis. A comparison of CPU and GPU implementations for solving the convection diffusion equation using the local modified SOR method. *Parallel Computing*, 40(7):173 – 185, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.
6. Yiannis Cotronis, Elias Konstantinidis, and Nikolaos M. Missirlis. *A GPU Implementation for Solving the Convection Diffusion Equation Using the Local Modified SOR Method*, pages 207–221. Springer International Publishing, Cham, 2014.
7. E. Konstantinidis and Y. Cotronis. A practical performance model for compute and memory bound GPU kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658, March 2015.
8. E. Konstantinidis and Y. Cotronis. A quantitative performance evaluation of fast on-chip memories of GPUs. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 448–455, Feb 2016.
9. Elias Konstantinidis and Yiannis Cotronis. Accelerating the red/black sor method using gpus with cuda. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I*, pages 589–598, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
10. Elias Konstantinidis and Yiannis Cotronis. Graphics processing unit acceleration of the red/black sor method. *Concurrency and Computation: Practice and Experience*, 25(8):1107–1120, 2013.
11. Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37 – 56, 2017.
12. Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

Advances in Possibilistic Clustering with Application to Hyperspectral Image Processing

S.D. Xenaki[*]

Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, GR-15236, Athens, Greece
Institute for Astronomy, Astrophysics, Space Applications and Remote Sensing, National Observatory of Athens, GR-15236, Penteli, Greece
`ixenaki@di.uoa.gr` or `ixenaki@noa.gr`

Abstract. Clustering is a well established data analysis methodology that has been extensively used in various fields of applications during the last decades. The main focus of the present thesis is on a well-known cost-function optimization-based family of clustering algorithms, called Possibilistic C-Means (PCM) algorithms. Specifically, the shortcomings of PCM algorithms are exposed and novel batch and online PCM schemes are proposed to cope with them. These schemes rely on (i) the adaptation of certain parameters which remain fixed during the execution of the original PCMs and (ii) the adoption of sparsity. The incorporation of these two characteristics renders the proposed schemes: (a) capable, in principle, to reveal the true number of physical clusters formed by the data, (b) capable to uncover the underlying clustering structure even in demanding cases, where the physical clusters are closely located to each other and/or have significant differences in their variances and/or densities, and (c) immune to the presence of noise and outliers. Moreover, theoretical results concerning the convergence of the proposed algorithms, also applicable to the classical PCMs, are provided. The potential of the proposed methods is demonstrated via extensive experimentation on both synthetic and real data sets. In addition, they have been successfully applied on the challenging problem of clustering in HyperSpectral Images (HSIs). Finally, a feature selection technique suitable for HSIs has also been developed.

1 Introduction and Related Work

Clustering is a well established data analysis methodology that lie in the framework of pattern recognition and it has been extensively used in various fields of applications during the last decades. Given a set of *objects*, the aim of clustering is the identification of groups (*clusters*) formed by "similar" objects. A great amount of work reported in the clustering literature has been devoted to the identification of compact and hyperellipsoidally shaped clusters. Each such

[*] Dissertation Advisor: Sergios Theodoridis, Professor

cluster is represented by a vector called *cluster representative* or simply *representative*, which lies in the same l-dimensional space with the data and (ideally) is located at the *center* of the cluster.

The most well-known algorithms that deal with this problem, belong to the family of cost optimization clustering algorithms and are the *k-means* (hard clustering), e.g. [1], the *fuzzy c-means* (FCM - fuzzy clustering), e.g. [2], [3] and the *possibilistic c-means* (PCM - possibilistic clustering), e.g. [4], [5], [6], [7], [8]. The main goal of all these algorithms is to move iteratively the representatives towards the centers of the regions that are *dense in data points* (dense regions), that is, to regions where significant aggregations of data points (clusters) exist. Under this perspective, we say that each such vector *represents* a cluster, while their movement towards the centers of the clusters is carried out via the minimization of appropriately defined cost functions.

Let us consider first the k-means and FCM, which share some significant features. First of all, they both require prior knowledge of the exact number of clusters m underlying in the data set (which, of course, is rarely known in practice). In addition, in both schemes the updating equations of the representatives are interrelated. As a result, these algorithms *impose* a specific clustering structure on the data set (rather than uncovering the underlying one), in the sense that they will return m clusters *irrespective* of the actual number of physical clusters existing in the data set. Specifically, if m is less than the actual number of clusters, at least some representatives will fail to move to dense regions, while in the opposite case, some naturally formed clusters will split into more than one pieces[1]. A common method for estimating m is via the use of suitable validity indices (e.g., [8]). Finally, as shown in [6], [7], k-means and FCM are vulnerable to noisy data and outliers.

As far as the PCM algorithms are concerned, the cluster representatives are updated, based on the *degrees of compatibility* of the data vectors with the clusters. In contrast to FCM and k-means, in PCM algorithms, the degrees of compatibility of a data vector with the various clusters are mutually independent. A direct consequence of this fact is that even if the number of clusters is overestimated, in principle, all representatives will be driven to dense regions, making thus feasible the uncovering of the actual clusters. However, in this case, the scenario where two or more cluster representatives are led to the same dense in data region, may arise, which, however, can be faced after the termination of the algorithm by seeking for (almost) coincident representatives. In addition, PCM deals well with noisy data points and outliers, compared to k-means and FCM. However, it involves additional parameters, usually denoted by γ. Each of these parameters is associated with a single cluster, while their accurate estimation is of crucial importance. Since, once they have been estimated they are kept fixed during the execution of the PCM algorithm, it is clear that poor initial estimates are likely to lead to poor clustering performance, especially in more

[1] Of course, if the value of m corresponds to the actual number of physical clusters, the algorithms have the ability to recover the physical clusters; that is, in this case "imposition" coincides with "uncovering".

demanding data sets (e.g. where clusters with significantly different variances are encountered in the data set).

2 Dissertation Summary

The present thesis focuses on the Possibilistic C-Means (PCM) algorithms. Specifically, exposing first their shortcomings, they are extended next, in order to overcome them. These extensions rely on the adoption of the parameter adaptivity and the sparsity concepts. In the sequel, the main contributions of the present thesis are briefly exposed.

First, a novel approach in the context of possibilistic clustering algorithms, named *Adaptive Possibilistic C-Means* (APCM) has been developed [9], [10]. APCM addresses several of the weaknesses of original PCM, by allowing the *adaptation* of some parameters that are characteristic to all PCM algorithms, during its execution. This is in contrast to classical PCM algorithms where these parameters, once they are set, they remain fixed. This characteristic of APCM gives rise to two new features that are not met in classical PCM algorithms. The first one is that APCM is capable, in principle, to reveal the true number of physical clusters, provided that it starts with a reasonable overestimate of it, thus overcoming a long-standing issue in the clustering literature. This is carried out by removing the clusters that gradually become obsolete (i.e., the clusters whose characteristic parameter diminishes towards zero as the algorithm evolves). The other feature resulting from the adaptation of the characteristic parameters of APCM is the increase of its flexibility in following the variations in the formation of the clusters during the algorithm execution. This makes APCM able to uncover the underlying clustering structure, even in demanding cases, where the physical clusters are closely located to each other and/or have significant differences in their variances. APCM is compared against several related state-of-the-art algorithms through extensive simulations on both synthetic and real data and the provided results show that APCM exhibits superior performance in almost all the considered data sets. Moreover, theoretical results that are indicative of the convergence behavior of the algorithm are also provided.

Next, we extended PCM by introducing the concept of *sparsity*. The rationale behind this extension is that, in practice, a data point is most compatible with at most one, a few or even none cluster (outlier). Thus, taking into account the data points that are most compatible with a given cluster and excluding those that are not compatible with it, leads to more accurate estimations of the clusters' parameters. The resulting algorithm, called *Sparse Possibilistic C-Means* (SPCM) [11] can deal well with closely located clusters that may also be of significantly different densities, while at the same time it exhibits immunity to noise and outliers. Finally, a non-trivial convergence proof for the SPCM algorithm is conducted [12]. The main source of difficulty in the provided convergence analysis, compared to those given for previous possibilistic algorithms, relies on the fact that one of its updating parameter equations is not given in closed form but is computed via a two-branch expression, which defines a non-continuous mapping. In this thesis, it is shown that SPCM will converge to one of the local minima of its associated cost function. As a side effect, it is shown that similar convergence results can be derived for the PCM algorithm, viewed as a special case of SPCM, which are stronger than those established in previous works.

In the sequel, the main features of the proposed APCM and SPCM algorithms are combined giving rise to the *Sparse Adaptive Possibilistic C-Means* (SAPCM) algorithm [13], [11], which, inheriting all the advantages of its ancestors, has the ability to (a) cope well with demanding data sets with closely located physical clusters with possibly different densities and/or variances, (b) determine the number of physical clusters and (c) improve even more the estimates of the clusters' parameters, compared to APCM and SPCM. Extensive experimentation verified the overall advantages of SAPCM compared to other related algorithms. Moreover, two variants of SAPCM, which use the above original SAPCM algorithm as a building block, have been devised. The first one is an iterative bottom-up version, called *Sequential SAPCM* (SeqSAPCM) [14], which, at each iteration, determines a single new cluster by employing SAPCM. Thus, it unravels sequentially the underlying clustering structure. The basic advantage of SeqSAPCM is that it does not require knowledge of the number of physical clusters (not even a crude overestimate, as is the case with APCM, SPCM and SAPCM). The second variant of SAPCM is called *Layered SAPCM* (L-SAPCM) [15] and works in layers. Specifically, the SAPCM algorithm is initially applied in the whole data set and then it is recursively applied individually on each resulting cluster, in order to reveal possible clustering structure within it, working in a tree structure basis. L-SAPCM terminates when none of the clusters resulting so far has further clustering structure within it. As is verified by the experimental results, L-SAPCM can provide accurate clustering even in cases where the data form closely located clusters at various "resolutions", i.e. the variances of the clusters may differ orders of magnitude from each other.

Also, a considerable contribution of this thesis is the development of an online version of the APCM algorithm, called *Online APCM* (O-APCM) [16], which processes data points one by one and memorizes their impact to suitably defined accumulating variables. O-APCM embodies three new procedures for (a) generating, (b) merging or (c) deleting clusters dynamically and it is a good candidate for clustering of big data sets, whose size and dimensionality are prohibitive for batch algorithms. Finally, it is highlighted that O-APCM may be utilized for applications in both stationary, as well as dynamically varying environments, where the physical clusters may change their location in data space over time. Specifically, O-APCM has the ability to weight more heavily the most recent data, compared to older data, in the estimation of its parameters. Experimental results show that O-APCM offers high discrimination ability at a very low computational cost for data sets in stationary conditions and, additionally, it is able to track with high accuracy the physical clusters at a non-stationary environment. Finally, the application of O-APCM to a real video data set, in order to identify and track moving objects, highlights its great potential in monitoring the evolution of dynamically varying phenomena.

The potential of the proposed methods is also demonstrated via experimentation on the basis of three case studies, concerning real hyperspectral images (HSIs). The images have been collected from different hyperspectral sensors and depict various land cover cases. The proposed algorithms gave, in general, superior performance compared to other related algorithms.

Finally, a sparsity-aware feature selection technique suitable for HSIs has been developed in the frame of the current thesis [17]. The proposed method is based on the optimization of a sparsity promoting cost-function, in order to identify the bands with the most significant ability in discriminating the various homogeneous regions in the HSI under study. Experimental results on real HSI data have shown remarkable quality of the clustering considering only the selected bands that result from the above technique.

3 Results and Discussion

In the sequel, we describe in detail one of the proposed possibilistic clustering algorithms, that is the Sparse Adaptive Possibilistic C-Means (SAPCM), that incorporates the idea of adaptivity and sparsity.

3.1 Sparse Adaptive Possibilistic C-Means Algorithm

The SAPCM algorithm stems from the optimization of the cost function

$$J(\Theta, U) = \sum_{j=1}^{m} \left[\sum_{i=1}^{N} u_{ij} \|\mathbf{x}_i - \boldsymbol{\theta}_j\|^2 + \gamma_j \sum_{i=1}^{N} (u_{ij} \ln u_{ij} - u_{ij}) \right] + \lambda \sum_{i=1}^{N} \|\mathbf{u}_i\|_p^p \quad (1)$$

where $u_{ij} > 0$, $i = 1, ..., N$, $j = 1, ..., m$, the parameter γ_j is related to the "size" of jth cluster, C_j, and it could be described as a measure of its variance around its $\boldsymbol{\theta}_j$, and λ is a parameter that controls the degree of the imposed sparsity.

In SAPCM, the parameters γ, after their initialization, are properly adapted as the algorithm evolves. In particular, the parameter γ of each specific cluster is updated based only on those data vectors that are "*most compatible*" with this cluster. The proposed SAPCM algorithm stems from the optimization of the cost function of eq. (1), by setting

$$\gamma_j = \frac{\hat{\eta}}{\alpha} \eta_j \quad (2)$$

with η_j being a measure of the mean absolute deviation of C_j as it has been formed in the current iteration, $\hat{\eta}$ is a constant defined as the minimum among all initial η_j's, i.e., $\hat{\eta} = \min_{j=1,\dots,m_{ini}} \eta_j$, where m_{ini} is the initial number of clusters, and α is a user-defined positive parameter, so that the ration $\hat{\eta}/\alpha$ approximates the mean absolute deviation of the smallest physical cluster. Note that although the latter quantity is fixed for a given data set, it is unknown in practice.

Initialization in SAPCM: First, we make an overestimation, denoted by m_{ini}, of the true number of natural clusters m, formed by the data points; that is, we begin with m_{ini} $\boldsymbol{\theta}_j$'s and their corresponding η_j's. Regarding $\boldsymbol{\theta}_j$'s and η_j's, their initialization drastically affects the final clustering result in SAPCM. Recalling that SAPCM is a possibilistic-type algorithm and these algorithms move the cluster representatives towards "dense in data points" regions (physical clusters), care should be taken so that at least one representative lies "close" to each physical cluster with its associated η_j being initialized suitably. Thus, a good starting point for them is of crucial importance. To this end, the initialization of $\boldsymbol{\theta}_j$'s is carried out using the final cluster representatives obtained from the FCM algorithm, when the latter is run with m_{ini} clusters. Taking into account that FCM is very likely to drive the representatives to dense in data regions (since $m_{ini} > m$), the probability that at least one of the initial $\boldsymbol{\theta}_j$'s is placed in each dense region (cluster) of the data set, increases with m_{ini}.

After the initialization of $\boldsymbol{\theta}_j$'s, η_j's are initialized as follows:

$$\eta_j = \frac{\sum_{i=1}^{N} u_{ij}^{FCM} \|\mathbf{x}_i - \boldsymbol{\theta}_j\|}{\sum_{i=1}^{N} u_{ij}^{FCM}}, \quad j = 1, \ldots, m_{ini}, \tag{3}$$

where $\boldsymbol{\theta}_j$'s and u_{ij}^{FCM}'s in eq. (3) are the final parameter estimates obtained by FCM. Combining eqs. (2), (3), the initialization of γ_j's is completely defined.

Parameter adaptation in SAPCM: In SAPCM algorithm, all parameters are adapted during its execution. More specifically, this refers to, (a) the parameters $\boldsymbol{\theta}_j$'s, (b) the parameters u_{ij}'s, (c) the number of clusters, m, and (d) the parameters γ_j's, with (c) and (d) being achieved through two interrelated processes. Minimization of $J(\Theta, U)$ with respect to $\boldsymbol{\theta}_j$ leads to the same updating equation as in the original PCM scheme, that is

$$\boldsymbol{\theta}_j = \frac{\sum_{i=1}^{N} u_{ij}\mathbf{x}_i}{\sum_{i=1}^{N} u_{ij}} \tag{4}$$

Taking the derivative of $J(\Theta, U)$ with respect to u_{ij}, we obtain

$$\frac{\partial J(\Theta, U)}{\partial u_{ij}} \equiv f(u_{ij}) = d_{ij} + \gamma_j \ln u_{ij} + \lambda p u_{ij}^{p-1}, \tag{5}$$

where $d_{ij} = \|\mathbf{x}_i - \boldsymbol{\theta}_j\|^2$. Obviously, $\frac{\partial J(\Theta,U)}{\partial u_{ij}} = 0$ is equivalent to $f(u_{ij}) = 0$, the solution of which will give the requested u_{ij}. Clearly, this equation cannot be solved analytically. However, it can be efficiently solved arithmetically based on the following propositions.

Proposition 1 *$f(u_{ij})$ does not become zero for $u_{ij} \in (-\infty, 0) \cup (1, +\infty)$* [2].

Proposition 2 *The stationary points of $f(u_{ij})$ are $\hat{u}_{ij} = \left[\frac{\lambda}{\gamma_j} p(1-p)\right]^{\frac{1}{1-p}}$ and $\tilde{u}_{ij} = +\infty$.*

[2] The proofs of Propositions 1 to 6 are given in the dissertation.

Proposition 3 *The unique minimum of $f(u_{ij})$ appears at $\hat{u}_{ij} = \left[\frac{\lambda}{\gamma_j} p(1-p)\right]^{\frac{1}{1-p}}$.*

Proposition 4 *If $f(\hat{u}_{ij}) < 0$ then $f(u_{ij}) = 0$ has exactly two solutions $u_{ij}^{\{1\}}$, $u_{ij}^{\{2\}} \in (0,1)$ with $u_{ij}^{\{1\}} < u_{ij}^{\{2\}}$.*

Proposition 5 *If $f(u_{ij}) = 0$ has two solutions $u_{ij}^{\{1\}}$, $u_{ij}^{\{2\}}$ (with $u_{ij}^{\{1\}} < u_{ij}^{\{2\}}$), $J_{SPCM}(\Theta, U)$ exhibits a local minimum at the largest of them ($u_{ij}^{\{2\}}$).*

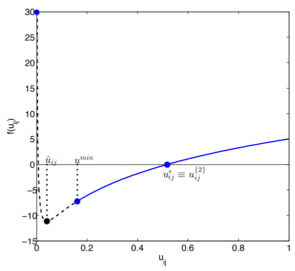
Proposition 6 *$J_{SPCM}(\Theta, U)$ exhibits its global minimum (with respect to u_{ij}) at u_{ij}^*, where:*

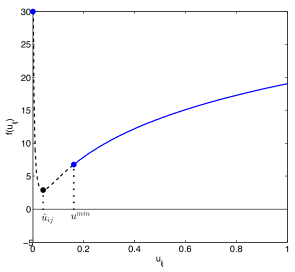
$$u_{ij}^* = \begin{cases} u_{ij}^{\{2\}}, \text{ if } f(\hat{u}_{ij}) < 0 \text{ and } u_{ij}^{\{2\}} > \left(\frac{\lambda(1-p)}{\gamma_j}\right)^{\frac{1}{1-p}} & (\equiv u_{min}) \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

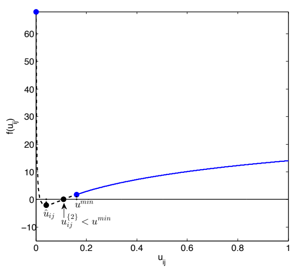
Based on the above propositions, to determine u_{ij}^*, we solve $f(u_{ij}) = 0$ as follows. First, we determine $\hat{u}_{ij}$ and check whether $f(\hat{u}_{ij}) > 0$. If this is the case, then $f(u_{ij})$ has no roots in $[0,1]$. Note that, in this case, it is $f(u_{ij}) > 0$ for all $u_{ij} \in (0,1]$, since $f(\hat{u}_{ij}) > 0$ (Fig. 1b). Thus, J_{SPCM} is increasing with respect to u_{ij} in $(0,1]$ (Fig. 1e). Consequently, in this case we set $u_{ij}^* = 0$, *imposing sparsity.* In the rare case, where $f(\hat{u}_{ij}) = 0$, we set $u_{ij}^* = 0$, as $\hat{u}_{ij}$ is the unique root of $f(u_{ij}) = 0$ and $f(u_{ij}) > 0$ for $u_{ij} \in (0, \hat{u}_{ij}) \cup (\hat{u}_{ij}, 1]$. If $f(\hat{u}_{ij}) < 0$, then $f(u_{ij}) = 0$ has exactly two solutions that both lie in $[0,1]$ (see Figs. 1a, 1c). In order to determine the largest of the solutions ($u_{ij}^{\{2\}}$), we apply the bisection method (e.g. [18]) in the range $(\hat{u}_{ij}, 1]$, as $u_{ij}^{\{2\}}$ is greater than $\hat{u}_{ij}$. The bisection method is known to converge very rapidly to the optimum u_{ij}, that is, in our case, to the largest of the two solutions of $f(u_{ij}) = 0$. If the obtained solution $u_{ij}^{\{2\}}$ satisfies the rightmost condition in the first branch of eq. (6), then we set $u_{ij}^* = u_{ij}^{\{2\}}$ (Fig. 1d) . Otherwise, u_{ij}^* is set to 0 (see Fig. 1f).

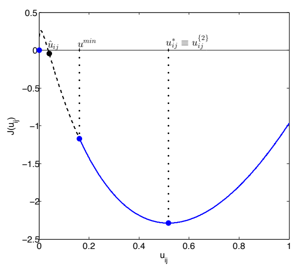
Concerning the adjustment of the number of clusters $m(t)$ at the tth iteration, we proceed as follows. Let *label* be a N-dimensional vector, whose ith element is the index of the cluster which is *most compatible* with $\mathbf{x}_i$, that is the index j for which $u_{ij}(t) = \max_{r=1,\ldots,m(t)} u_{ir}(t)$. At each iteration of the algorithm, the adjustment (reduction) of the number of clusters $m(t)$ is achieved by examining, for each cluster C_j, if its index j appears at least once in the vector *label* (i.e. if there exists at least one vector $\mathbf{x}_i$ that is most compatible with C_j). If this is the case, C_j is preserved. Otherwise, C_j is eliminated and, thus, U and Θ are updated accordingly. As a result, the current number of clusters $m(t)$ is reduced.

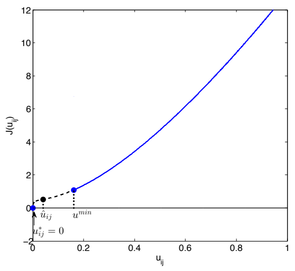
Finally, concerning γ_j's and in contrast to the classical PCM where they are kept fixed, in SAPCM they are given by eq. (2) and are *adapted* at each iteration of the algorithm through the adaptation of the corresponding η_j's. More specifically, we propose to compute the parameter η_j of a cluster C_j at


(a) $f(u_{ij})$


(b) $f(u_{ij})$


(c) $f(u_{ij})$


(d) $J(u_{ij})$


(e) $J(u_{ij})$

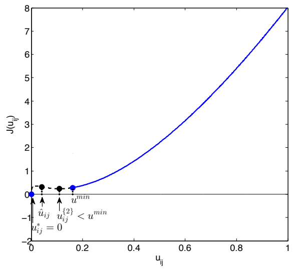

(f) $J(u_{ij})$

Fig. 1: In all plots the dashed parts of the graphs correspond to the interval $(0, u_{min})$, which is not accessible by the algorithm (see eq. (6)). (a) The shape of function $f(u_{ij})$, when $f(\hat{u}_{ij}) < 0$ and the right-most condition of eq. (6) is satisfied and (d) the corresponding shape of the cost function $J(u_{ij})$. (b) The shape of function $f(u_{ij})$, when $f(\hat{u}_{ij}) > 0$ and (e) the corresponding shape of $J(u_{ij})$. (c) The shape of function $f(u_{ij})$, when $f(\hat{u}_{ij}) < 0$ and the right-most condition of eq. (6) is not satisfied and (f) the corresponding shape of $J(u_{ij})$.

each iteration, as the *mean absolute deviation* of the most compatible to cluster C_j data vectors, i.e.,

$$\eta_j(t+1) = \frac{1}{n_j(t)} \sum\nolimits_{\mathbf{x}_i : u_{ij}(t) = \max_{r=1,\ldots,m(t+1)} u_{ir}(t)} \|\mathbf{x}_i - \boldsymbol{\mu}_j(t)\|, \tag{7}$$

where $n_j(t)$ denotes the number of the data points $\mathbf{x}_i$ that are most compatible with C_j at iteration t and $\boldsymbol{\mu}_j(t)$ the mean vector of these data points.

Selection of parameter $\boldsymbol{\lambda}$: As it follows from the previous analysis, considering a specific data point $\mathbf{x}_i$ and a cluster C_j, a necessary condition in order for the equation $f(u_{ij}) = 0$ to have a solution is $f(\hat{u}_{ij}) < 0$, which, taking into account eq. (5) and solving with respect to λ gives $\lambda < \frac{\gamma_j}{p(1-p)} \exp\left(-1 - \frac{d_{ij}(1-p)}{\gamma_j}\right)$. Consequently, selecting

$$\lambda \geq \frac{\gamma_j}{p(1-p)} \exp\left(-1 - \frac{d_{ij}(1-p)}{\gamma_j}\right), \tag{8}$$

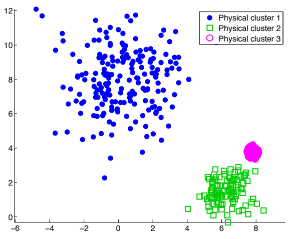
the degree of compatibility u_{ij} of a data point $\mathbf{x}_i$ with a cluster C_j is set to 0, promoting sparsity. Aiming at retaining the smallest sized cluster, say C_q (i.e., the cluster with $\gamma_q = \min_{j=1,\dots,m} \gamma_j$) until the termination of the algorithm (provided of course that at least one representative has been initially placed in it), a reasonable choice for λ would be the one for which u_{ij} becomes 0 for points $\mathbf{x}_i$ that lie at distance d_{iq} greater than γ_q from the representative $\boldsymbol{\theta}_q$. In this way, $\boldsymbol{\theta}_q$ will be less likely to be "attracted" by nearby larger clusters, aiding it to remain in the region of the physical cluster where it was first placed. This is so because the cluster representative will be affected only by the data points that are very close to it (i.e., points with $d_{iq} < \gamma_q = \min_{j=1,\dots,m} \gamma_j$).

To this end, applying inequality (8) for d_{ij} and γ_j equal to $\gamma_q = \min_{j=1,\dots,m} \gamma_j$, we end up with $\lambda \geq \frac{\gamma_q}{p(1-p)\mathrm{e}^{2-p}}$, where e is the base of natural logarithm. In practice, we select λ as

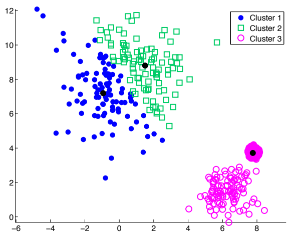
$$\lambda = K \frac{\min_{j=1,\dots,m} \gamma_j}{p(1-p)\mathrm{e}^{2-p}}, \tag{9}$$

where if we set $K = 1$, we allow non-zero u_{ij}'s for points that lie at distance around γ_q from $\boldsymbol{\theta}_q$. In most of the experiments of SAPCM, we take $K = 0.1$.

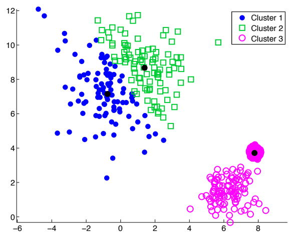
Comparison of APCM with state-of-the-art clustering algorithms In this section, we compare the clustering performance of SAPCM with that of the k-means, the FCM, the PCM [5], the UPC [8], the UPFC [19], the PFCM [7], the SPCM-L_1 [20], the APCM [10] and the SPCM [11] algorithms, which all result from cost optimization schemes. For a fair comparison, the representatives $\boldsymbol{\theta}_j$'s of all algorithms (except for SPCM-L_1) are initialized based on the FCM scheme and the parameters of each algorithm are first fine tuned. Moreover, in PCM, UPC, UPFC, PFCM and SPCM, duplicate clusters are removed after their termination. In order to compare a clustering with the true data label information, we utilize (a) the Success Rate (SR) of each physical cluster ($\mathrm{SR}_{c_j}, j = 1, ..., m$), which measures the percentage of the points of each physical cluster that have been correctly labeled by each algorithm, (b) the mean of the Euclidean distances (MD) between the true mean of each physical cluster and its closest cluster representative obtained by each algorithm, (c) the number of iterations (Iter) and (d) the total time required (Time) for the convergence of each algorithm. **Experiment**: Consider a two-dimensional data set consisting of $N = 5300$ points, where three clusters C_1, C_2 and C_3 are formed. Each cluster is modelled by a normal distribution. The means of the distributions are $\mathbf{c}_1 = [0.27, 7.99]^T$, $\mathbf{c}_2 = [6.28, 1.49]^T$ and $\mathbf{c}_3 = [7.81, 3.76]^T$, respectively, while their covariance matrices are set to $3 \cdot I_2$, $0.5 \cdot I_2$ and $0.01 \cdot I_2$, respectively. A number of 200 points are generated by the first distribution, 100 points are generated by the second one and 5000 points are generated by the third one. Note that C_2 and C_3 clusters are very close to each other and they have a big difference in their variances (see Fig. 2a). Also, note the difference in the density among the three clusters.



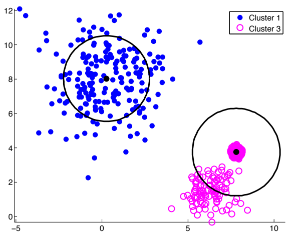
(a) The data set



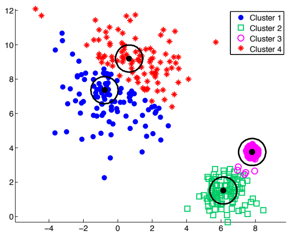
(b) k-means



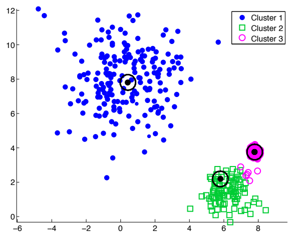
(c) FCM



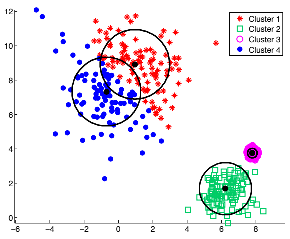
(d) PCM



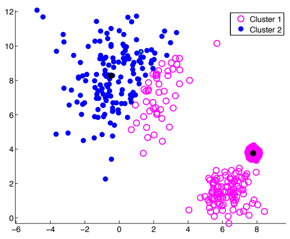
(e) UPC



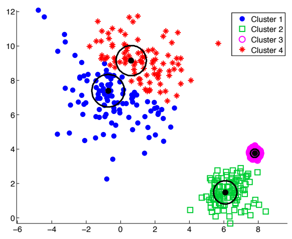
(f) UPFC



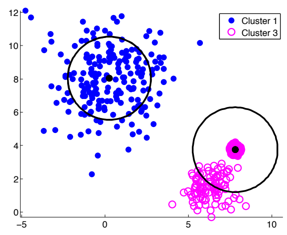
(g) PFCM



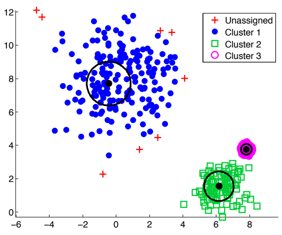
(h) SPCM-L$_1$



(i) APCM



(j) SPCM



(k) SAPCM

Fig. 2: (a) The data set of Experiment. Clustering results for (b) k-means, $m_{ini} = 3$, (c) FCM, $m_{ini} = 3$, (d) PCM, $m_{ini} = 5$, (e) UPC, $m_{ini} = 5$, $q = 1.5$, (f) UPFC, $m_{ini} = 10$, $\alpha = 5$, $\beta = 1$, $q = 2.2$, $n = 3$, (g) PFCM, $m_{ini} = 5$, $K = 1$, $\alpha = 1$, $\beta = 5$, $q = 1.5$, $n = 1.5$, (h) SPCM-L_1, $\lambda = 15$, $q = 2$ (i) APCM, $m_{ini} = 5$, $\alpha = 0.3$, (j) SPCM, $m_{ini} = 5$, and (k) SAPCM, $m_{ini} = 10$ and $\alpha = 0.15$.

Table 1 shows the results of all algorithms for Experiment. Fig. 2b and Fig. 2c show the clustering obtained using the k-means and FCM algorithms, respec-

Table 1: Performance of clustering algorithms for the data set of Experiment.

	m_{ini}	m_{final}	SR$_{c_1}$	SR$_{c_2}$	SR$_{c_3}$	MD	Iter	Time
k-means	3	3	51	0	100	3.4066	2	0.265
k-means	5	5	51	94	51.48	0.5369	20	0.202
FCM	3	3	51	0	100	3.3432	110	0.140
FCM	5	5	50.50	93	51.62	0.5537	86	0.218
PCM	5	2	100	0	100	0.9242	15	0.514
PCM	10	2	100	0	100	0.9254	18	1.185
UPC ($q = 1.5$)	5	4	50	95	100	0.4589	65	0.390
UPC ($q = 1.2$)	10	4	50	95	100	0.4480	89	0.910
UPFC ($a = 5$, $b = 1$, $q = 2$, $n = 1.5$)	5	4	50.50	96	100	0.4170	41	0.390
UPFC ($a = 5$, $b = 1$, $q = 2.2$, $n = 3$)	10	3	100	94	100	0.3601	190	2.940
PFCM ($K = 1$, $a = 1$, $b = 5$, $q = 1.5$, $n = 1.5$)	5	4	51.50	100	100	0.4573	38	0.380
PFCM ($K = 1$, $a = 2$, $b = 1$, $q = 2$, $n = 1.2$)	10	5	44	97	100	0.4011	60	0.880
SPCM-L$_1$ ($\lambda = 15$, $q = 2$)	-	2	76	0	100	1.1831	6	0.031
APCM ($\alpha = 0.3$)	5	4	53	100	100	0.4469	73	0.390
APCM ($\alpha = 0.3$)	10	4	52.50	100	100	0.4748	90	0.889
SPCM ($K = 0.9$)	5	2	100	0	100	0.9256	15	3.276
SPCM ($K = 0.9$)	10	2	100	0	100	0.9263	19	7.769
SAPCM ($\alpha = 0.18$)	5	3	**100**	**100**	**100**	**0.3222**	91	13.40
SAPCM ($\alpha = 0.15$)	10	3	**100**	**100**	**100**	**0.3020**	100	18.94

tively, both for $m_{ini} = 3$. Figs. 2d, 2e, 2f, 2g, 2h, 2i and 2j, depict the performance of PCM, UPC, UPFC, PFCM, SPCM-L_1, APCM and SPCM, respectively, with their parameters chosen (after fine-tuning) as stated in the caption. In addition, the circles, centered at each $\boldsymbol{\theta}_j$ and having radius $\sqrt{\gamma_j}$ (as they have been computed after the convergence of the algorithms), are also drawn.

As it can be deduced from Fig. 2 and Table. 1, even when the k-means and the FCM are initialized with the (unknown in practice) true number of clusters ($m = 3$), they fail to unravel the underlying clustering structure mainly due to the big difference in the variances and densities between clusters. The classical PCM also fails to detect the physical cluster 2, because it is located very close to the densest physical cluster. The UPC algorithm has been fine tuned so that the parameters γ_j's, which remain fixed during its execution and are the same for all clusters, get small enough values, in order to identify cluster C_2. However, it splits the high variance/low density cluster C_1 in two clusters. The same seems to hold for the PFCM algorithm, after fine tuning of its several parameters. The UPFC algorithm produces 3 clusters, at the cost of a computationally demanding fine tuning of the (several) parameters it involves. The APCM algorithm also splits the big variance cluster in two subclusters, failing to detect the underlying clustering structure. On the other hand, SPCM identifies two clusters with high accuracy with respect to the center of the actual clusters, but misses the third one. Finally, as it is deduced from Table 1, the SAPCM algorithm manages to identify all clusters, achieving the best SR and MD results and estimating very accurately the true centers of the clusters, since it exhibits the minimum MD among all algorithms.

References

1. J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-Means Clustering Algorithm", *Journal of the Royal Statistical Society*, vol. 28, pp. 100-108, 1979.
2. J. C. Bezdek, "A Convergence Theorem for the Fuzzy Isodata Clustering Algorithms", *IEEE Trans. on Pattern Analysis and Mach. Intel.*, vol. 2, pp. 1-8, 1980
3. J. C. Bezdek, "Pattern Recognition with Fuzzy Objective Function Algorithms", *Plenum*, 1981
4. R. Krishnapuram and J. M. Keller, "A Possibilistic Approach to Clustering", *IEEE Transactions on Fuzzy Systems*, vol. 1, pp. 98-110, 1993.
5. R. Krishnapuram and J. M. Keller, "The Possibilistic C-Means Algorithm: Insights and Recommendations", *IEEE Trans. on Fuzzy Systems*, vol. 4, pp. 385-393, 1996.
6. S. Theodoridis and K. Koutroumbas, "Pattern Recognition", *Academic Press*, 2009.
7. N. R. Pal and K. Pal and J. M. Keller and J. C. Bezdek, "A Possibilistic Fuzzy C-Means Clustering Algorithm", *IEEE Trans. on Fuzzy Systems*, pp. 517-530, 2005
8. M. S. Yang and K. L. Wu, "Unsupervised Possibilistic Clustering", *Pattern Recognition*, vol. 39, pp. 5-21, 2006.
9. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "Adaptive Possibilistic Clustering", *IEEE International Symposium on Signal Processing and Information Technology*, pp. 422-427, Dec 2013.
10. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "A Novel Adaptive Possibilistic Clustering Algorithm", *IEEE Transactions on Fuzzy Systems*, vol. 24, no. 4, pp. 791-810, 2016.
11. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "Sparsity-aware Possibilistic Clustering Algorithms", *IEEE Transactions on Fuzzy Systems*, vol. 24, no. 6, pp. 1611-1626, 2016.
12. K. D. Koutroumbas and S. D. Xenaki and A. A. Rontogiannis, "On the Convergence of the Sparse Possibilistic C-Means Algorithm", *IEEE Transactions on Fuzzy Systems*, 2017, to appear. DOI: 10.1109/TFUZZ.2017.2659739
13. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "Sparse Adaptive Possibilistic Clustering", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 3072-3076, Florence 2014.
14. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "Sequential Sparse Adaptive Possibilistic Clustering", *SETN 2014: Artificial Intelligence: Methods and Applications*, pp. 29-42, 2014.
15. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis and O. A. Sykioti, "A Layered Sparse Adaptive Possibilistic Approach for Hyperspectral Image Clustering", *IEEE Intern.Geo. and Rem.Sens.Sympos(IGARSS)*, pp. 2890-2893, 2014.
16. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis, "Hyperspectral Image Clustering Using a Novel Efficient Online Possibilistic Algorithm", *24th European Signal Processing Conference (EUSIPCO)*, pp. 2020-2024, 2016.
17. S. D. Xenaki and K. D. Koutroumbas and A. A. Rontogiannis and O. A. Sykioti, "A New Sparsity-Aware Feature Selection Method for Hyperspectral Image Clustering", *IEEE Intern. Geoscience and Rem. Sens. Symposium (IGARSS)*, pp. 445-448, 2015.
18. G. Corliss, "Which Root Does the Bisection Algorithm Find?", *Siam Review*, vol. 19, pp. 325-327, 1977.
19. X. Wu, B. Wu, J. Sun, H. Fu, "Unsupervised Possibilistic Fuzzy Clustering", *Journal of Information & Computational Science*, vol. 5, pp. 1075-1080, 2010.
20. Y. Hamasuna and Y. Endo, "On Sparse Possibilistic Clustering with Crispness Classification Function and Sequential Extraction", *Soft Computing and Intel. Syst. (SCIS) and 13th Intern. Symp. on Advanced Intel. Syst. (ISIS)*, pp. 1801-1806, 2012.