Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
http://w3.ele.tue.nl/nl/

Series title:
Master graduation paper,
Electrical Engineering

Commissioned by Professor:
prof.dr. H. Corporaal

Group / Chair:
Electronic Systems

Date of final presentation:
January 12, 2012

Report number:

-

Quantifying and Capturing the Semantics of Computational Problems in Contemporary Applications for Algorithmic Choice

by

Author: R. Jongerius

Internal supervisor: prof.dr. H. Corporaal

External supervisor: P. Stanley-Marbell, Ph.D.

Disclaimer
The Department of Electrical Engineering of the Eindhoven University of Technology accepts no reponisibility for the contents of M.Sc. theses or practical training reports

Where innovation starts

Quantifying and Capturing the Semantics of Computational Problems in Contemporary Applications for Algorithmic Choice

Rik Jongerius, IBM Research — Zürich
r.jongerius@ieee.org

***Abstract*—Characterization and analysis of workloads in computing systems has traditionally focused on *algorithms* and their embodiment in applications. However, algorithms are just a means to the end goal of solving *computational problems*. Recently, new programming languages and system software platforms have been developed which provide performance trade-offs through the exploitation of *algorithmic choice*. Such techniques can be enabled by shifting the traditional focus on algorithms to computational problems.**

The dominant computational problems contained in a suite of 21 real-world applications are studied to quantify the potential for algorithmic choice as a means to gain performance improvements. It is demonstrated that 55 % of the aggregate, analyzed execution time of the applications is spent in a set of 16 compute problems. The properties of applications, making them amenable to separation of compute problems from algorithms, are discussed.

Results of the quantification showed that not all compute problems can be identified by a proper *name*. As a result, these difficult-to-name problems are not part of the 16 identified compute problems. In order for difficult-to-name problems to benefit from algorithmic choice, the Copernicus problem language is introduced as a means to identify the problems solved in applications. The Copernicus language is a declarative language aimed at capturing the actual *semantics* of compute problems, as opposed to their algorithmic implementations.

The language is analyzed by implementing 13 of the 16 previously identified compute problems, showing the capability of the language to capture real-world problems. Several of the descriptions in Copernicus are compared in size to their respective algorithmic implementations, demonstrating that the overhead of using these descriptions to identify algorithms solving identical compute problems is not an issue.

I. Introduction

Over the last four decades, computing systems have witnessed performance growth through a variety of *performance growth vehicles*. Performance increases in many of the early computing systems were achieved through the use of increasingly more complex instruction set architectures, with the goal of performing more work in hardware. These architectures were ultimately limited by the difficulty of compiling general-purpose applications to complex instruction sets [1], [2], [3].

As an alternative to complex instruction set computing (CISC) platforms, reduced instruction set computing (RISC) focused on simplified hardware, easier targeting of compilers, and the possibility of higher clock frequencies as the cycle time of (pipelined) simplified hardware reduced. Together with advances in semiconductor process technology, RISC pushed performance further during the following two decades. In the last few years, however, power dissipation and heat removal challenges, related to ever-higher clock frequencies, became unmanageable.

Recently, there has been a shift towards obtaining performance through parallelism [4]. Scaling through parallelism is typically classified as strong scaling, if performance is gained when problem sizes remain fixed and hardware concurrency is increased. In contrast, the term weak scaling refers to situations when increases in hardware concurrency alone are insufficient and performance is only gained when the problem sizes are increased appropriately to expose more data-level parallelism. Strong scaling will ultimately be limited by Amdahl's law [5], while weak scaling will be limited by the available data-level parallelism in contemporary data sets.

All of the aforementioned performance growth vehicles—complex hardware or hardware acceleration, increasing clock speeds, and parallelism—have the same fundamental goal: making (fixed) algorithms execute faster. In principle, however, computing systems are intended to solve computational problems, which are conceptually different from specific algorithms or implementations thereof in applications. By describing all necessary steps, algorithms specify *how* a computational problem can be solved.

Exposing the semantics of computational problems to hardware, the operating system, or to language runtimes opens up new opportunities for performance growth and program analysis. Instead of focusing on algorithms, focusing on computational problems enables algorithmic choice: the possibility for a runtime system to change the algorithmic implementation—depending on target execution architecture, input data sets and current system parameters—to achieve best performance, for example in terms of speed or energy-efficiency. Work by Ansel et al. [6] shows that significant performance benefits can be gained for a selected set of problems and associated algorithms.

The main driving force behind this work is to enable algorithmic choice by specifying the semantics of computational problems in a machine-readable form. The work presented addresses three issues related to algorithmic choice and the shift of focus from algorithms to computational problems:

- The feasibility of identifying well-defined computational problems, occupying a significant portion of execution time, in existing complex software applications. It is conjectured that, if such well-defined components can

be identified even in *existing* software, then providing a platform for algorithmic choice is likely of benefit.

- The types of applications which lend themselves to separation of their constituent problem definitions from algorithms, and which properties might prevent algorithmic choice to be of benefit.
- The issue of capturing the semantics of computational problems, independent of their algorithms, for which a language (named Copernicus) is introduced. Problem descriptions written in this language can be added as annotation to source code or inserted into binaries, and can be used to uniquely identify problems to facilitate algorithmic choice.

Following the discussion of relevant related work in Section II, Section III introduces the notion of computational problems independent of their algorithmic implementations. A quantitative study of the dominant computational problems in contemporary applications is presented and discussed in Section IV. The Copernicus problem language is introduced in Section V and used in Section VI to implement the previously identified dominant computational problems. Section VII concludes the paper and discusses future work.

II. Related Work

Traditionally, workload characterization has involved analyzing the properties of compiled applications, together with input data sets, executing on real hardware platforms such as shared-memory multiprocessors [7] or simulated hardware platforms such as microarchitectural simulators [8]. To quantify the potential of algorithmic choice as a performance growth vehicle, it is desired to have an understanding of the constituent compute problems in contemporary applications.

For a platform to benefit from algorithmic choice, a supporting framework is required. Recently, several frameworks have been introduced in literature. An example of such a framework is PetaBricks [6], which offers a programming language with the ability to specify multiple execution paths to solve a single computational problem. The PetaBricks autotuner performs a design space exploration over these user-supplied implementations, using the resulting information at run time to determine the best-performing method to solve a problem of a particular size. In a similar way, the elastic computing framework [9], provides an autotuner and code library containing multiple algorithms per computational problem. The characterization in this work provides a quantitative analysis of the potential applicability of such frameworks.

In contrast to the aforementioned general-purpose frameworks, FFTW [10] is a domain specific framework for algorithmic choice which tunes fast Fourier transform (FFT) algorithms to a target platform. The SPIRAL [11] framework permits the programmer to capture the semantics of linear signal transforms in the digital signal processing domain, which are then optimized and converted into code targeted to a given platform. Both FFTW and SPIRAL are, however, domain-specific.

In the elastic computing framework, compute problems are identified by a function's name. By calling the appropriately named routine, an application programmer specifies which problem needs solving. The downside of using such static names will be addressed later in this paper and is one of the motivations to introduce the Copernicus problem language to capture the actual semantics of compute problems.

Traditional, imperative programming languages specify *how* a compute problem is solved on a Von Neumann computer; the notion of program state and updating this state are important aspects of these type of languages [12], [13]. By using control-flow and assignment statements, imperative languages give a step-by-step approach to solve compute problems, and are thus a good match to implement algorithms.

In contrast to imperative languages, declarative programming languages focus on giving formulas, possibly in the form of predicate logic, defining properties on the data elements in programs. Declarative programs specify rather *what* has to be done, instead of how to get there; there are no explicit control-flow statements. In this sense, Copernicus is a declarative language as it aims to capture relations between variables without specifying any sequencing of statements.

Declarative languages such as Prolog, or the closely related class of functional languages such as Haskell, have no explicit control flow, however, they usually permit the programmer to use recursion. As a result, these languages allow implementation of algorithms, as recursion implicitly introduces sequencing into programs. Numerous examples of implementations of divide-and-conquer algorithms (e.g., QUICKSORT) can be found as these are easily implemented using recursion. As a result, Copernicus forbids the use of recursion. As work by Immerman [14] showed that languages without recursion are still expressive enough to describe a broad class of problems, the absence of recursion is not considered to be too restrictive.

Just as declarative languages, the experimental programming language UNITY [15] is a language without control flow. The programmer specifies assignment statements, which are repeatedly executed in random order until the program reaches a fixed point—the output does not change if any of the statements is executed. While there is no explicit control flow, sequencing of statements is part of the programming model and, as a result, the language implements algorithms.

A closely related language to Copernicus is the low-level Set assembly language (Sal) [16], which captures relations between data elements in the form of predicate trees. There are no constructs for recursion, and the set of operators is comparable to Copernicus. Sal can also be used to capture the semantics of computational problems, but Copernicus provides additional statements to capture the structure of input and, especially, output of problems, which is required for algorithmic choice. Furthermore, Copernicus is aimed to be a higher level language and is more programmer-friendly by introducing syntactic sugar to improve its human readability.

III. Computational Problems

In order to reason about *computational problems (CPs)*, consider the example of the graph partitioning problem. This problem can be solved using the well-known algorithm by Kernighan and Lin [17]. The algorithm has various different application areas, examples include electronic design

Listing 1. The computational problem of integer sort, expressed in the Copernicus language. Note that the problem description contains no hints about how to compute the output.

```
domain ::
 N : int<32>
 x : int<32>[N]
range ::
 y : int<32>[N]
relation ::
 n : int<32>
 exists y {
  forall n from 0 to N-2 { y[n] <= y[n+1] } and y >=< x
 } ;
```

automation, where it can solve the "placement" problem, or image processing, where it can be used to solve the "image segmentation" problem.

This example shows that a single algorithm can be used to solve different problems, each identified by a different *name*. However, the algorithm solving each problem is the same, indicating that there is apparently some commonality between these problems; they share common computational *semantics*. This commonality can be captured by introducing a formal definition to describe CPs, and is a motivation in itself to introduce a notation as described in Section V to capture the actual semantics.

For the remainder of this paper, a computational problem will be used to refer to the semantic properties of a computation, which determine what relation exists between the inputs and outputs of computation, as opposed to how that desired relation is achieved. This definition abstracts from implementation-specific details, such as algorithms, sequencing of instructions, used instruction set architectures, etc.

Definition 1. *Computational Problem (CP). A computational problem, CP(S_D, S_R, $\mathcal{R}$) is a 3-tuple representing an input set S_D and output set S_R, related by the relation $\mathcal{R}$.* ♢

S_D is the set of possible inputs of interest, or the *domain*, and S_R is the set of possible valid outputs of interest, or the *range* or co-domain; the relation $\mathcal{R}$ defines both what outputs are considered valid, as well as the relation between one or more of the inputs and a valid output. Intuitively, a CP specifies what the inputs and outputs of a portion of an application are, and what properties or invariants are satisfied by the inputs paired with the outputs.

To give an example of a CP described according to Definition 1, Listing 1 shows the integer sort compute problem in the Copernicus language (the language itself will be discussed in more detail in Section V). The problem is divided in three parts, representing the 3-tuple *CP(S_D, S_R, $\mathcal{R}$)*. The relation itself states that the output elements should be ordered (`y[n] <= y[n+1]`) and that the output `y` is a permutation of the input `x` (denoted by the Copernicus permutation operator, `>=<`).

Computational problems themselves do not imply the use of specific algorithms, as the relation between inputs and outputs could be achieved by a variety of algorithms. This results in a design space for CPs, as illustrated in Figure 1. During the

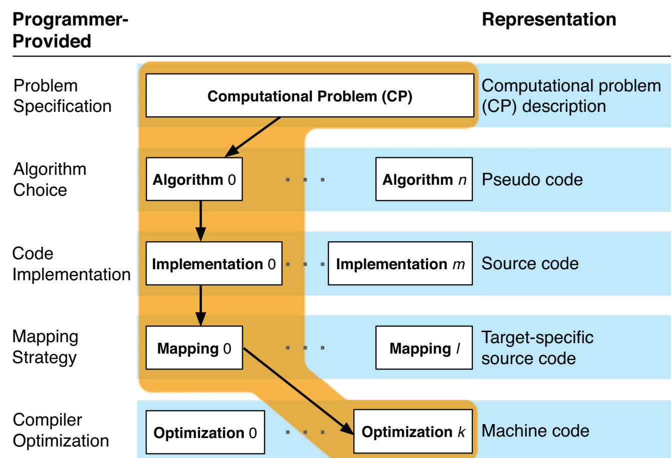

Figure 1. A computational problem can be solved using one or more different algorithms, and for each algorithm, different implementations can be created. Each implementation can then be mapped to different platforms and compiled using different compiler optimizations.

course of implementation of an application, the programmer makes a variety of architectural choices before the final binary is available for execution. Multiple algorithms may exist for solving a given CP; in implementing a given algorithm, there may exist several choices, e.g., for data structures. The source code corresponding to a high-level language implementation can be customized (mapped) to different platforms or different subsets of platforms (e.g., using architecture-specific intrinsics such as vector extensions), which leads to targeted source code. The targeted source code can finally be compiled using several different compiler optimizations.

Each of the decisions in the levels of the hierarchy below the CP, shown in Figure 1, do not affect the semantics of an application, but often have significant effects on performance or energy-efficiency. Understanding the constituent CPs in applications, and having a means to capture the semantics, may open up future opportunities for runtime systems or hardware to achieve improvements in compute performance without a need for re-implementation or even re-compilation of applications. The characterization of CPs in applications presented in Section IV shows the potential opportunities for implementation of such techniques.

IV. Quantification of Computational Problems

In order to quantify the presence of CPs in real-world applications that can be expressed in terms of Definition 1, a set of applications from a diverse range of domains was studied [18].

A. Methodology

The applications used in the study were taken from the SPEC CPU2006 [19] and MiBench [20] benchmark suites. The rationale for using these well-known benchmark suites is that the applications in these suites have already been selected as being representative of contemporary computing workloads in both the desktop / server market (SPEC) and the embedded / low-power computing market (MiBench). Since some of

Table I
APPLICATIONS FROM THE SPEC CPU2006 AND MIBENCH SUITES USED IN STUDY OF CPS IN CONTEMPORARY APPLICATIONS.

Benchmark	Domain	Dominant Computation
SPEC CPU2006		
400.perlbench (PRLB)	Programming language	Integer
401.bzip2 (BZIP2)	Compression	Integer
403.gcc (GCC2K6)	Compiler	Integer
429.mcf (MCF)	Combinatorial optimization	Integer
445.gobmk (GOBMK)	Artificial intelligence	Integer
456.hmmer (HMMR)	DNA pattern search	Integer
458.sjeng (SJENG)	Artificial intelligence	Integer
462.libquantum (LIBQ)	Physics (quantum computing)	Integer
464.h264ref (H264)	Video compression	Integer
471.omnetpp (OMPP)	Discrete event simulator	Integer
473.astar (ASTR)	Path finding	Integer
433.milc (MILC)	Quantum chromodynamics	Float
453.povray (POVRAY)	Computer visualization	Float
470.lbm (LBM)	Computational fluid dynamics	Float
482.sphinx3 (SPX3)	Speech recognition	Float
MiBench		
JPEG encode (JPGe)	Image compression	Integer
JPEG decode (JPGd)	Image compression	Integer
Rijndael encode (RIJNe)	Cryptography	Integer
Rijndael decode (RIJNd)	Cryptography	Integer
Susan (SUS)	Image processing	Integer
Lame (LAME)	Audio compression	Float

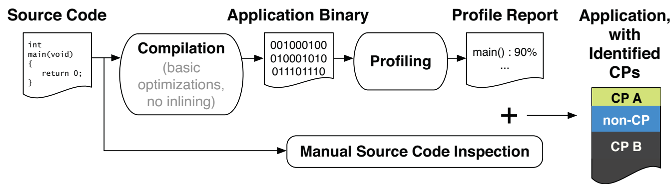

Figure 2. Methodology used to identify computational problems in applications. The applications were compiled and profiled, and the results from profiling together with manual code inspection were used to identify the CPs.

the applications in the MiBench suite are better regarded as kernels rather than full applications, only larger applications from the suite were employed. The 21 applications studied are listed in Table I, along with the general application domain and dominant type of arithmetic operations.

The quantification of constituent CPs in the applications under study ultimately requires manual analysis and understanding of source code. Since the applications in some cases comprise hundreds of thousands of lines of code, an approach, illustrated in Figure 2, was needed to make the analysis manageable. The applications were compiled with gcc, and profiled using gprof and OProfile in order to select the top five subroutines in terms of execution time for each application. Profiling also supplied execution time breakdowns on the program-statement and loop-nest levels. The resulting set of subroutines was subjected to manual source code inspection to identify the constituent CPs.

B. Results

The methodology was applied to the set of applications in Table I. For the profiling steps, the applications were compiled with optimization flags `-O1 -g`; higher levels of optimization

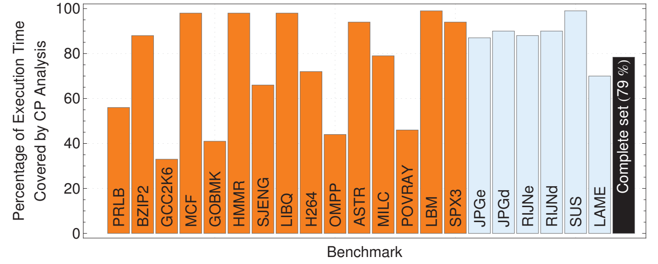

Figure 3. Percentage of execution time analyzed per application on an Atom D510. For example, the source code analyzed for PRLB accounts for 56 % of the execution time of that application.

were not employed to ensure that the obtained profiles were in close correspondence with the structure of the original source code, and that subroutines were not inlined.

The total set of selected subroutines, identified by the profiling step, covered 79 % of the aggregate execution time of the complete set of applications. Figure 3 lists the percentage of execution time covered by the analysis per application. For several applications (e.g., 470.lbm and Susan) up to 99 % of the execution time was covered by the analysis. The smallest fraction of execution time covered occured in the case of 403.gcc, where the analyzed source code only accounted for 33 % of the application's execution time. In general, for all applications containing a large number of subroutines the total execution time consumed by the top five subroutines was lower than for applications with a small number of subroutines. The fraction of execution time covered by the analysis can be increased by selecting more subroutines per application, however, no new subroutines where added as the analysis already covered a significant fraction of execution time.

Identified computational problems

Table II lists 16 of the CPs identified during the process of manual code inspection. Listed with each CP are the applications in which it occurs, the CP's duration of execution, and the percentage of the per-application execution time taken by the CP. In Table II, all CPs related to matrix arithmetic are grouped under "Matrix Operations" for brevity of exposition, even though matrix multiplication, matrix addition, etc. are indeed distinct CPs.

Figure 4 shows the fraction of *analyzed* execution time covered by the identified CPs in the complete set of applications. The DCT, DFT, and move-to-front transform are condensed in one slice for clarity of the figure. In total, the set of 16 identified CPs covers approximately 55 % of the total analyzed execution time of all applications.

Partial and full sorting of data, minimizing and maximizing search, and Fourier-related transforms (DCT and DFT) are the most common CPs across the suite in terms of the number of applications they can be found in, appearing in 5, 7, and 5 different applications respectively. Searching and the Fourier-related transforms, however, only account for a small fraction of execution time, while sorting and partial sorting account for at least 8 % of the total execution time.

In terms of execution time, CPs related to matrix arithmetic and the computational problem of finding the Viterbi path

Table II
COMPUTATIONAL PROBLEMS IDENTIFIED IN THE SET OF APPLICATIONS FROM THE SPEC CPU2006 AND MIBENCH BENCHMARK SUITES.

Computational Problem	Application	Execution Time (s)	% of Application
Regular expression matching	400.perlbench	1065	49.0
Sorting	401.bzip2	2006	64.7
	429.mcf	43	2.0
	445.gobmk	20	0.9
Partial sorting	453.povray	18	1.1
	471.omnetpp	450	24.4
Move-to-front transform	401.bzip2	313	10.1
Search	403.gcc	122	8.1
	429.mcf	46	2.2
	445.gobmk	94	4.1
	473.astar	445	20.6
Maximizing search	458.sjeng	131	4.7
	482.sphinx3	65	1.4
Minimizing search	464.h264ref	374	9.5
Minimum-cost network flow	429.mcf	1776	83.8
Matrix operations	433.milc	1576	63.5
	470.lbm	2339	62.3
Finding Viterbi path	456.hmmer	3189	96.1
	482.sphinx3	61	1.3
Sum of absolute differences	464.h264ref	1307	33.3
DCT-II	464.h264ref	25	0.7
	JPEG encode	0.008	11.2
DCT-III	464.h264ref	21	0.5
	JPEG decode	0.003	12.3
DCT-IV	Lame	0.155	10.0
DFT	482.sphinx3	58	1.3
	Lame	0.327	21.2
Mahalanobis distance	482.sphinx3	2071	45.2

account for the largest fractions, at 12.3 % and 10.2 %, of the aggregate analyzed execution time of the set of applications. Both problems are responsible for a large fraction of execution time in only a restricted set of two applications.

Regular expression matching, the move-to-front transform, finding the minimum-cost network flow, the sum of absolute distances, and the Mahalanobis distance only occur in single applications. While these CPs might seem to be of less interest, they individually still account for up to 4 % of the total execution time of applications. Moreover, we can still envision them to be common to many applications. A straightforward example is regular expression matching. This CP is not limited to 400.perlbench, but is also used in, for example, network packet analysis. The move-to-front transform can be applicable to multiple compression algorithms; similar arguments can be made for the other CPs.

Categorization of applications

Additional insights where gained during the process of manual inspection of source code. The difficulty of identifying the constituent CPs in Table II depends on the structuring of the application in question and programming style. In some cases it was relatively straightforward to identify the computational problem being solved independent of the specific algorithm implemented in the source code; in a few cases neither the algorithm nor the computational problem solved were easily determined.

Based on properties of both the source code and the executed binaries, the applications can be divided into the four categories listed in Table III. The amount of effort required to identify the constituent CPs varies across the categories. If these identified CPs are also intended candidates for algorithmic replacement, then it is desirable to have well-defined boundaries within a program (e.g., subroutines) at which such replacement may take place; this is also captured in the construction of the four categories.

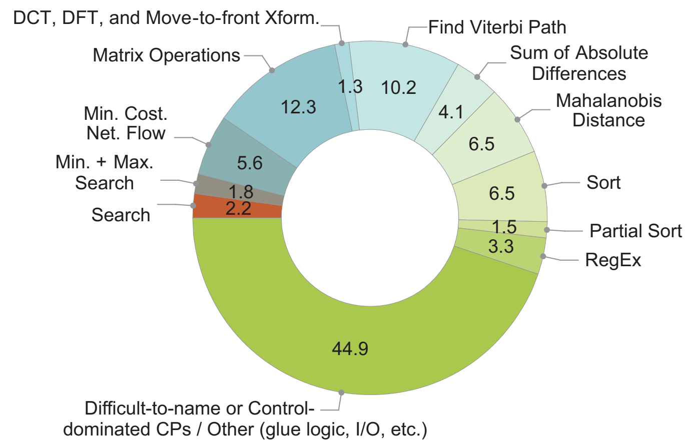

Figure 4. Percentage of aggregate analyzed execution time of the complete set of applications covered by the identified computational problems.

CP category 1: single CP per procedure

The applications in this category are structured such that each procedure solves an isolated sub-problem. With code inspection, these sub-problems could easily be identified as well-defined CPs. One example of an application in this category is 433.milc. The top five subroutines which dominate execution time each implement an isolated CP, in this case a CP related to matrix arithmetic. There is, for example, a subroutine which performs matrix-matrix multiplication, while another subroutine performs matrix addition.

CP category 2: multiple CPs per procedure

Several applications have compound procedures which solve multiple CPs consecutively. Other applications solve a single, well-defined CP in a procedure with significant regions of additional glue logic, such as initialization of data structures, file I/O, and so on; following Definition 1, this glue logic is not considered part of the CP. In the applications belonging to this category however, it is still possible to identify one or more isolated CPs in a given subroutine. For example, in two of the analyzed subroutines in 464.h264ref, several problems are solved consecutively: first a 2-dimensional discrete cosine transform of type II (2D DCT-II) is performed on the data, followed by a quantization of the result, and, finally, the computation of the 2D DCT-III of the transformed and quantized data.

CP category 3: multiple procedures per single CP

In this category, CPs are implemented by combinations of several procedures, making it difficult to identify the CP solved by a single procedure. For example, in 429.mcf, several of its procedures do not solve easily-named CPs. However, when looking at multiple procedures, it becomes apparent that together they solve a minimum-cost network flow problem.

Table III
CATEGORIZATION OF APPLICATIONS BASED ON PROGRAM STRUCTURE AND TYPE OF COMPUTATIONAL PROBLEMS SOLVED. CATEGORIES 2 AND 3 BOTH CONTAIN 401.BZIP2 SINCE THE COMPRESSION AND DECOMPRESSION PARTS OF THE SOURCE CODE ARE SIGNIFICANTLY DIFFERENT IN STRUCTURE.

Category 1 (Single CP per procedure)	**Category 2** (Multiple CPs per procedure)	**Category 3** (Multiple procedures per single CP)	**Category 4** (Difficult-to-name or control-dominated CPs)
433.milc	401.bzip2	400.perlbench	403.gcc
453.povray	464.h264ref	401.bzip2	445.gobmk
456.hmmer	Lame	429.mcf	458.sjeng
470.lbm	Susan	473.astar	462.libquantum
483.sphinx3			471.omnetpp
JPEG encode			Rijndael encode
JPEG decode			Rijndael decode

CP category 4: difficult-to-name or control-dominated CPs

There are multiple applications which have procedures for which it is difficult to identify the problem being solved; such applications account for about a third of the applications studied. Several of the procedures have an irregular control-dominated structure for which it is expected that there is not much (if any) algorithmic choice. An example is 458.sjeng; its procedures are control-dominated and specific to the semantics of a chess-playing engine. Other applications solve difficult-to-name problems: problems for which it is difficult to give a short, descriptive name.

Instruction mix breakdown

In order to verify the categorization of applications in category 4, Figure 5 shows the instruction mix (branches, loads, stores, floating-point, and other instruction types) in the applications studied. The instruction mix was obtained by profiling the applications on an Intel Atom D510 platform using the available hardware performance counters. It is observed that applications falling in the first two categories of Table III also typically have below-average fractions of branch instructions, while applications in categories 3 and 4 have above-average fractions of branches. For category 4, this indeed indicates that the respective applications are control dominated, as was observed from the manual source code inspection. As the CPs identified in category 3 are large and more complex, they are naturally split across multiple procedures; it is not surprising that they on average have more branch / control instructions.

There are, however, two exceptions. The JPEG encoder and decoder (JPGe and JPGd) have, according to Figure 5, above-average fractions of control instructions, but are classified in category 1. The higher fraction of control instructions in this case is related to the Huffman coding step in JPEG. Similarly, for the Rijndael encoding and decoding applications (RIJNe and RIJNd), there is a below-average fraction of control instructions, even though the applications are classified in category 4. In this case, their classification in category 4 is not because of control dominance, but rather that the applications implement difficult-to-name CPs.

C. Conclusions on quantification

The potential gain in performance from algorithmic choice depends on the fraction of execution time for which CPs can be identified. Applications with a high fraction of execution time covered by CPs, can potentially gain more than applications with a low fraction. A programming style where the CPs have well-defined boundaries at which the replacement can take place, such as the style of applications in categories 1 and 3, are a likely match for algorithmic choice. Approximately half of the applications fall in these two categories.

In total, 55 % of the total analyzed execution time of the 21 applications was identified as 16 easily-named CPs. For these CPs, there may exist algorithm variants that expose performance, power dissipation or energy-efficiency tradeoffs when compared across different implementations or executed on different architectures. As the applications in question cover a broad range of real-world applications, this gives credence to the hypothesis that there is an opportunity for frameworks improving performance by algorithmic choice.

The question arises if algorithmic choice can still be of benefit for the remaining 45 % of the analyzed execution time, not identified as named CPs in Table II. The categorization of applications shows that applications in categories 2 and 4 contain glue logic or control-dominated sections of code, where algorithmic choice will likely not be of benefit. However, in category 4, several applications also contain difficult-to-name CPs. It is possible to describe these problems in terms of Definition 1, however, it is difficult to give a short, descriptive name, and, as such, they are not part of the 16 identified CPs. For these CPs, algorithmic choice can still be of benefit if there is a different method, other than naming, to uniquely identify problems. The Copernicus problem language is introduced next as a means to describe these difficult-to-name CPs.

V. COPERNICUS COMPUTATIONAL PROBLEM LANGUAGE

As discussed in Sections III and IV, there is need for a notation or language designed to uniquely identify CPs independent of algorithms implementing them. This section introduces the Copernicus problem language. Instead of giving CPs a name, which can obfuscate potential commonality or can be cumbersome to use if there is no clear name available, the notation captures the actual *semantics* of the computational problem according to Definition 1.

The main intention of the language is to enable algorithmic choice. However, the potential use of the language is much broader, and five different application areas are identified.

Algorithmic choice. Code sections in executable binaries can be annotated with a description of the CP in the Copernicus language. A runtime system can use the annotated CP description to identify sections of code in other libraries or binaries

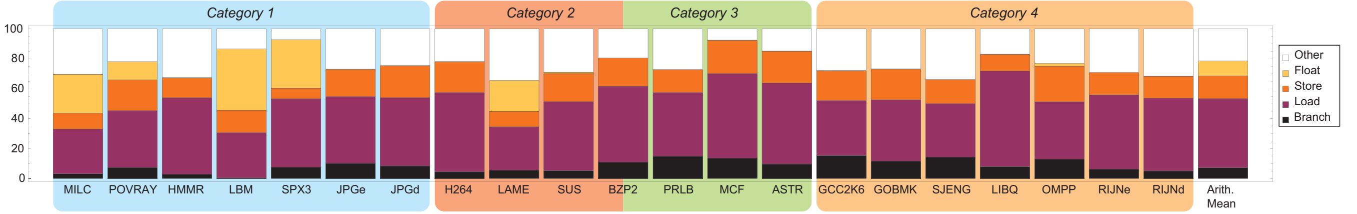

Figure 5. Instruction mix breakdown on an Intel Atom D510. Applications from categories 3 and 4 in Table III generally have high fractions of control instructions, whereas applications in categories 1 and 2 have lower fractions of control instructions.

solving the same problem, and, if allowed, replace one by the other if there is potential for performance improvement. The choice of whether or not to replace can be based on prior performance measurements of both implementations. Using a semantic description instead of a name to identify sections of code solving equal problems potentially enlarges the pool of code in applications eligible for algorithmic choice.

Documentation. The semantic description of the CP can be used as documentation of algorithms. It rather describes what the associated code sections do, instead of how the (possibly highly optimized) algorithms solve the problem.

Problem analysis. There are potential multiple analyses which can be performed if a semantic description is available. A first example is to perform an analysis of the computational complexity of the problem. Work by Immerman [14] shows that the descriptive complexity of a language is related to the computational complexity of problems which can be described in that same language. The descriptive complexity is determined by the available operators, or, to which class of logic a language belongs to (e.g., first- or second-order logic). By quantifying the operators used in a CP description, it is possible to determine what the most restrictive logic class is containing these operators, and thus to determine the computational complexity of the problem.

A second example of a possible analysis is to make statements about inherent parallelism in the computational problem. This can be done based on an analysis of data dependencies between input and output of the problem.

Verification. The description fully captures the semantics of the CP an algorithm implements and can therefore be used to verify correct operation of the algorithm. A potential approach is to execute the algorithm given some input, and verify if the corresponding produced output satisfies the relation in the problem description.

Problem solving. There is in principle sufficient information available in problem descriptions to actually solve the problem. There are two possible approaches for solving a Copernicus problem. The first method is to use search heuristics such as the Set virtual machine (Svm) [16] uses to solve Set assembly language (Sal) programs. A second, future method would be to automatically synthesize the semantic description into a software algorithm or hardware accelerator. Domain-specific methods as SPIRAL [11] already exist, but are not yet generally applicable.

A. *Copernicus Language Design*

Several aspects have influenced the design of the Copernicus problem language. To facilitate an algorithmic choice framework, the language has to provide a means to uniquely identify computational problems independent of the algorithms implementing them. Such a framework has several practical issues which the language has to account for. Furthermore, the language is designed to be used by programmers, and contains elements to aid them in their programming tasks. A detailed description of the language can be found in the language reference manual [21].

According to Definition 1, the notions of domain S_D, the input set, range S_R, the output set, and relation $\mathcal{R}$ have been made explicit in the language. The set of operators is restricted such that it is non-trivial to express algorithms in the language; there are no operators implying control flow or any form of sequencing of statements. As an example, recursion is not available as it would allow implementation of many divide-and-conquer algorithms. However, in order for the language to be sufficiently expressive, the set of operators was selected to capture at least second-order logic, as Immerman [22] showed that this can describe at least the problems in the polynomial-time hierarchy (PH).

Recall the computational problem of integer sort, shown in Listing 1, Section III. The problem definition is divided into three parts using the `domain`, `range`, and `relation` keywords. These keywords note the start of the respective sections describing the three elements of Definition 1.

An implementation of algorithmic choice replaces algorithms on well-defined boundaries in code, for example between a function's `call` and `return` instructions. As a result, the language should not only capture the domain, range, and relation between input and output, but also more practical issues such as in-memory representations of data.

Therefore, Copernicus has support for several basic data types, such as `int`, `real`, or `nat` (any natural number including 0). From these basic data types, structures or multi-dimensional matrices can be constructed. The combination of data type, explicit data size in bits and structure form the *universe* of the domain and range of the CP.

The integer sort example in Listing 1 uses two arrays as input and output for data, plus an additional variable indicating the size of those arrays. All elements are 32 bits wide, such that each can represent values from -2^{31} to $2^{31} - 1$.

While the description of the universes in a problem captures the structure of the in-memory representation of domain and range variables, this information is not sufficient to determine if two algorithmic implementations can be replaced with each

Table IV
AVAILABLE VARIABLE-BINDING OPERATORS IN COPERNICUS AND THE RESPECTIVE MATHEMATICAL INTERPRETATION.

Copernicus operator	Mathematical representation
`forall x { p(x) }`	$\forall x\, p(x)$
`exists x { p(x) }`	$\exists x\, p(x)$
`sum x from 0 to N { f(x) }`	$\sum_{x=0}^{N} f(x)$
`prod x from 0 to N { f(x) }`	$\prod_{x=0}^{N} f(x)$
`min for x { f(x) }`	$\min_{x} f(x)$
`max for x with x > 0 { f(x) }`	$\max_{x>0} f(x)$

other; information regarding the actual location of the data is still missing. Therefore, there is the need for a separately provided *mapping table*, which maps domain and range variables to physical memory or register locations in the hardware platform.

Syntactic sugar was added to make the language human readable. Key elements are its support for arithmetic functions and Boolean predicates to structure problem definitions. Together with the basic arithmetic and logical operators, several operators where added to simplify programming; an example is the permutation operator >=<. Furthermore, there are several quantifiers and variable-binding operators available, listed in Table IV together with their respective mathematical representations. The `from` / `to` and `with` statements can be used with any variable-binding operator and determine the range of the operator.

The relational sentence in Listing 1 starts with the `exists` quantifier, and consists of two parts. One is the requirement that the output array `y` should be ordered and the other that `y` is a permutation of the input array `x`.

B. Descriptive complexity

As will be informally shown below, the operators of Copernicus are selected such that they implement at least second-order logic (SOL). Using SOL, it is possible to capture all problems in complexity class PH. Computational complexity theory [14] shows that

$$P \subseteq NP \subseteq PH.$$

The Copernicus language is thus capable of describing at least all problems in complexity classes P and NP, making it possible to describe a large set of problems to be generally applicable.

In order to verify that Copernicus is indeed at least a SOL, consider a variable x over a universe of natural numbers defined as

$$x \in \mathbb{N} \wedge (0 \leq x < 20).$$

Furthermore, there is a set $\mathbf{S}$ of elements over the same universe and a logic predicate $p(x)$. If the second order existential (SOE) sentence

$$\exists \mathbf{S} \forall x \,(x \in \mathbf{S} \vee p(x)) \tag{1}$$

can be expressed in the Copernicus language, then the operators in Copernicus describe at least a SOE language.

Listing 2. Linear programming in the Copernicus problem language.

```
domain ::
 N : int<32>     //Number of elements in x
 M : int<32>     //Number of problem constraints
 A : int<32>[M,N]
 b : int<32>[M]
 c : int<32>[N]
range ::
 x : int<32>[N]
relation ::
 cost : int<32>            //Temporary cost variable
 i : int<32> = <0 to N-1>  //Temporary variable
 j : int<32> = <0 to M-1>  //Temporary variable

 //Problem constraints
 fun constraint(y : int<32>[N]) : bool<1> ::=
  forall j { sum i { a[j,i] * y[i]} <= b[j] };

 //Non-negativity constraint
 fun positivity(y : int<32>[N]) : bool<1> ::=
  forall i { y[i] >= 0 } ;

 //Objective function
 exists cost {
  cost == max for x with constraint(x) and positivity(x)
   { sum i { c[i] * x[i] }
 } };
```

The set $\mathbf{S}$ can be modeled using an array with a maximum length of the size of the universe, such that it can uniquely contain all elements in the universe. The variable x is modeled as a scalar. The required variables can be declared as:

```
N : int<32> = <0 to 20>
S : int<32>[N] = <0 to 19>
x : int<32> = <0 to 19>
```

The first line declares `N` to be a 32 bit variable of type integer. The range of values `N` can represent is restricted between 0 and 20. An array `S` of `N` elements, each a 32 bit integer, is declared, which can take values between 0 and 19. Note that, as the size of `N` is at most 20, the array `S` has at most 20 elements. The last line declares the variable `x`.

In order to express the sentence in Equation 1 in Copernicus, an additional variable `n` is introduced as an index variable. The sentence can now be captured in Copernicus.

```
n : int<32> = <0 to N-1>
exists N { exists S {
  forall x { exists n { S[n] == x } or p(x) }
} };
```

The same argument can be made for any second-order universal (SOU) sentence, informally showing that the operators in Copernicus can indeed capture at least SOE and SOU, and thus SOL sentences.

C. Example

A more elaborate example of a problem description in the Copernicus language is given in Listing 2. The problem being captured is linear programming, which can be expressed as

$$\begin{aligned} &\text{maximize } \mathbf{c}^{\top}\mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b} \\ &\text{and } \mathbf{x} \geq 0. \end{aligned}$$

In this example, the vector $\mathbf{x}$ is considered the output of the problem. The relation itself consists of the declaration of several additional variables on lines 10 to 12, two Boolean

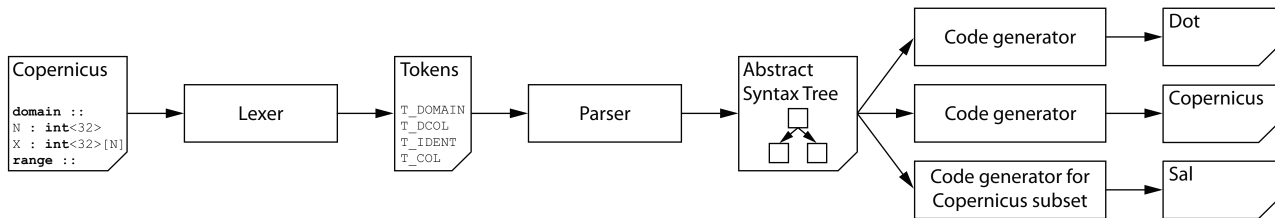

Figure 6. Copernicus compiler tool flow. The compiler is used to verify correctness of Copernicus implementations. Three code generators are used for verification; the abstract syntax tree can be visualized using dot, outputted to a Copernicus problem or compiled to the Set assembly language (Sal) [16].

predicates starting on lines 15 and 19 and the objective function starting on line 23. The two Boolean predicates show the use of functions as a means to structure a Copernicus problem. The variable `x` is subjected to several constraints in the predicates `constraint(...)` and `positivity(...)`. Both predicates could have been inlined into the objective function, however this would not be beneficial for readability.

VI. Language Analysis

As the Copernicus language is designed for algorithmic choice, problem definitions in the language are used to uniquely identify algorithms solving the same CP. In order to quantify the usability of the language for real-world problems and to verify its expressiveness, 13 CPs from Table II in Section IV were implemented in the language.

For algorithmic choice, a description of the CP is added to the code sections in binaries in order to identify the problem solved. These descriptions should, preferably, not be too large in size as to reduce the overhead of the added description. The size of the problem descriptions in bytes is used as a quality metric for the language. Other metrics such as execution time are meaningless as the performance of an application is based on the algorithms accompanied with the problem definition.

The problem description written by the programmer in the Copernicus language will be compiled to an optimized format, designed to expose as much potential commonality between problems as possible, before it is annotated into binaries. Determining equality of two CP descriptions based on the Copernicus language is made difficult by the syntactic elements added to aid human-readability of the problem; programmers can chose their own variable and function names, or structure a problem differently. Therefore, the problem description is to be compiled in order to eliminate these programmer-related properties.

Currently, such an optimized format for annotating problems is not yet available. In order to approximate the size of the annotated description, it is possible to calculate the size of the in-memory representation of the problem—the abstract syntax tree and supporting data structures—in the compiler. However, the in-memory representation is unnecessarily large (mainly because of the use of several 32-bit pointers and data values to build the tree) to be used for annotation.

The approximate size of the description is therefore based on a third available representation: the string of tokens accepted by the parser to generate the in-memory representation. It is known that there are 58 different tokens, which can be encoded easily using 1 byte per token. Furthermore, identifiers (variable names) are encoded using an additional byte; this limits a program to use at most 256 unique identifiers. Since the largest problem implemented uses only 39 unique identifiers, this is considered sufficient. Finally, constants are encoded in 4 bytes. Using these numbers, the size in bytes of the string of tokens can be calculated.

The compiler tool chain for Copernicus is shown in Figure 6. A Copernicus problem is first transformed by the lexer into a string of tokens. This string of tokens is parsed, and the abstract syntax tree is generated. A Copernicus problem is valid if it is accepted by the parser. Correctness of the problem itself can be verified by generating code from the abstract syntax tree and evaluating the output. In determining the size of the Copernicus problems, the problem is first parsed and output in Copernicus format is generated. This allows the compiler to eliminate programmer-related artifacts and to perform possible optimizations (e.g., constant folding or other algebraic optimizations) on the problem description in an attempt to expose as much potential commonality between problems for algorithmic choice. The resulting problem is fed back into the lexer to generate the string of tokens which represents the compiled format.

A. Problem implementation

Out of the 16 identified CPs, three problems were selected for a detailed analysis; most of the remaining 13 CPs were also implemented and are discussed later. For the three selected problems, an implementation of the problem in the Copernicus language has been made which closely matches the actual semantics in the applications. Data structures used by the algorithms were implemented in Copernicus and the exact semantics of the algorithms captured.

Aimed to span simple, moderately complex, and complex CPs, the three selected problems are the *complex matrix multiplication* (CMMUL), the *2D discrete cosine transform, type II* (2D DCT-II), and the *minimum-cost network flow* (MCF). The three problems are implemented using a small, medium, and large sized algorithm in the applications. Several of their implementation details are discussed below.

Complex matrix multiplication. The application 433.milc performs several complex matrix multiplications. The subroutine requires two matrices as input and one matrix as output, which are 3×3 matrices of a data structure of two doubles for the real and imaginary parts of the complex numbers. The implementation of these data types in Copernicus is straightforward, and the implementation is given in Listing 3. Note that by adding an additional domain variable `N` and replacing

Listing 3. Complex matrix multiplication CP from 433.milc in Copernicus.

```
typedef ::
 complex : struct {
  r : real<64>
  i : real<64> }
domain ::
 A : complex[3,3]
 B : complex[3,3]
range ::
 C : complex[3,3]
relation ::
 n,m,k : int<32> = <0 to 2>

 fun real_complex_mul(a,b : complex) : real<64> ::=
    a.r * b.r - a.i * b.i ;
 fun imag_complex_mul(a,b : complex) : real<64> ::=
    a.r * b.i + a.i * b.r ;

 exists C { forall n,m {
  C[n,m].r == sum k { real_complex_mul(A[n,k], B[k,m]) }
  and
  C[n,m].i == sum k { imag_complex_mul(A[n,k], B[k,m]) }
 } } ;
```

all constants 3 and 2 by `N` and `N-1` respectively, the CP can be generalized to an $N \times N$ matrix multiplication.

2D Discrete cosine transform, type II. The 2D DCT-II is used by two applications, but the implementation in jpeg-encode was chosen as the reference. The implementation calculates an in-place 2D DCT-II of an 8×8 matrix of floating-point numbers. The problem in Copernicus requires a distinct domain and range, so, in the problem description, two matrices are defined. The fact that the calculations in the algorithm are performed in-place can be captured by mapping both matrices to the same physical location using the previously mentioned mapping table. The implementation is not given in this article, but can be found in the language reference manual [21].

Minimum-cost network flow. In 429.mcf, the MCF problem is solved for vehicle scheduling. The problem is much larger than the matrix multiplication or the 2D DCT-II, and uses large data structures for input and output. As the language does not yet support pointer operations, all data structures are modeled as arrays and all pointers converted to indices into arrays. An implementation of the problem is given in Listing 4.

While implementing the domain and range of the problem, two issues were encountered. First, the data structures both have member variables which are part of the domain, and member variables which are part of the range. The same trick as with the 2D DCT-II is possible here, except that it requires to implement relational equality constraints for all static member variables; variables whose values are not updated by the algorithm (otherwise, if the algorithm is replaced with a second one which does update these variables, it might break consecutive code sections). This adds several additional relational statements which are not part of the actual problem, and therefore the choice was made to create separate data structures for the range variables. Secondly, the data structures contain additional variables which are used by the algorithm, but are not required by the problem itself. These additional variables are, however, part of the structures implemented in the CP, as they are required to capture the layout of data in memory. Listing 4 does not show these variables, but they are counted in the size of the problem description.

Listing 4. Minimum-cost network flow CP from 429.mcf, member variables not used by the relation are removed for brevity of the code.

```
typedef ::
 MCF_arc : struct {
  tail : int<32>        //Index of tail node
  head : int<32>        //Index of head node
  cost : real<64>       //Cost of arc
  upper : real<64>      //Flow upper bound of arc
  lower : real<64> }    //Flow lower bound of arc
 MCF_node : struct {
  balance : real<64> }  //Supply/demand of the node
 MCF_network : struct {
  N : int<32>           //Number of nodes
  M : int<32>           //Number of arcs
  nodes : MCF_node[N]   //Input nodes
  arcs : MCF_arc[M] }   //Input arcs
domain ::
 network : MCF_network
range ::
 feasible : bool<1>
 flow : real<64>[network.M] //Flow from MCF_arc
 optcost : real<64>         //Cost from MCF_network
relation ::
 i : int<32> = <0 to network.N-1>
 k : int<32> = <0 to network.M-1>
 a : int<32>

 //Flow conservation constraint:
 fun flow_cons(flow : real<64>[network.M]) : bool<1> ::=
   forall i { network.nodes[i].balance ==
     sum k with network.arcs[k].head == i { flow[k] } -
     sum k with network.arcs[k].tail == i { flow[k] }
 } ;

 //Flow capacities:
 fun flow_capacities(flow : real<64>[network.M]) : bool
      <1> ::= forall k { network.arcs[k].lower <= flow[k]
       and flow[k] <= network.arcs[k].upper } ;

 //Objective function
 exists feasible { feasible == exists optcost { optcost
      == min for flow with flow_cons(flow) and
      flow_capacities(flow) { sum k { network.arcs[k].
      cost * flow[k] } } } } ;
```

The two issues are highly related, and can partly be accounted for by the programmer when implementing a new algorithm together with a CP description. It is not difficult to make different data structures for input and output in both the CP and the algorithmic implementation, however, it may be more difficult to remove the additional variables, not used by the problem, from the domain. In the MCF problem, the structure of the network is passed to the algorithm in multiple (redundant) encodings. One is sufficient for the CP, but the others may be needed for an efficient algorithmic implementation.

B. Problem analysis

Table V lists the sizes of the compiled implementation of each of the three computational problems and the sizes of the compiled algorithmic implementations for the Intel IA-32 and IBM POWER7 instruction sets[1]. The table shows that, depending on the density of the instruction sets and the problem itself, the compiled Copernicus problem description is 1.3 to 6.1 times smaller in size than the algorithmic implementation.

[1]The size of the MCF algorithm is an approximation since, during the execution of the algorithm, several temporary results are stored for later use to improve performance of other calculations. This is not captured by the CP description in Copernicus.

Table V
COMPARISON OF THE APPROXIMATE SIZE OF THREE COMPILED COPERNICUS PROBLEMS WITH THE SIZE OF THE RESPECTIVE ALGORITHMS COMPILED TO TWO INSTRUCTION SETS.

CP	Compiled problem representation	Intel IA-32	IBM POWER7
CMMUL	391 B	507 B	848 B
2D DCT-II	345 B	965 B	1520 B
MCF	704 B	2,802 B	4336 B

Table VI
APPROXIMATE SIZES OF THE COMPILED COPERNICUS PROBLEM DESCRIPTIONS AND FRACTIONS COVERED BY THE DOMAIN AND RANGE FOR 13 OF THE CPS IDENTIFIED IN SECTION IV.

Computational Problem	Compiled problem (bytes)	Fraction covered by domain and range
Sorting	154	47 %
Search	108	43 %
Max. Search	150	43 %
Min. Search	161	54 %
MCF	704	51 %
CMMUL	391	25 %
Finding Viterbi path	421	33 %
Sum of abs. differences	129	55 %
2D DCT-II	345	24 %
2D DCT-III	393	21 %
1D DCT-IV	211	36 %
DFT	513	24 %
Mahalanobis distance	279	37 %

The algorithmic implementation of the 2D DCT-II is almost twice as large as the implementation of CMMUL, while the compiled problem definition of CMMUL is slightly larger than the 2D DCT-II. Apparently, the problems can be described in almost identical size, but an efficient implementation of the 2D DCT-II is larger. Furthermore, the relative overhead of a CP description decreases as the algorithmic size increases. The descriptions of the relation between input and output do not increase as fast as the algorithmic implementations; writing down the steps needed to solve a problem requires increasingly more space compared to writing down the problem itself.

Most of the other computational problems identified in Table II have been implemented in Copernicus. For these computational problems, the approximated sizes of the descriptions in the compiled format are listed in Table VI. The sizes of the algorithmic implementations are not given since the CP implementations are not defined using the data structures used in the applications, or the algorithmic implementations are not easily isolated in the applications' source code. As a result, a comparison would not be legitimate.

The 13 computational problems implemented in Copernicus, require between 108 and 704 bytes for the compiled description. The sizes of the smaller CP of sorting and search variants are dominated by the description of the required domain, range, and relation variables. For the larger CPs, the variable declarations have a less dominating role as the description of the more complex relational statements increases in size. An interesting exception is MCF, this large CP description is also dominated by variable declarations. The most important reason is that the MCF algorithm requires several input variables which are captured in the problem's domain, but which are not required by the problem's relation.

Three remaining computational problems have not yet been implemented. Regular expression matching is a difficult problem, and is essentially a programming language, making the CP description complex. Both partial sorting and the move-to-front transform are easily solved using recursion. As Copernicus does not allow recursion, the problems have to be expressed using other operators. Whether this is possible depends on the computational complexity of the problems. The problem of finding the Viterbi path is also easily solved using recursion, but it was no problem to implement it in Copernicus.

VII. CONCLUSION AND FUTURE WORK

The work presented in this article started with a quantification of *computational problems (CPs)* found in contemporary applications. CPs capture the *semantics* of compute problems independent of specific algorithms for implementing those computations. An understanding of the compute problems in applications is of interest in order to quantify the potential for performance benefits by employing *algorithmic choice* frameworks—the substitution of one algorithm solving a given CP, by another solving the same CP. The insights from quantification revealed the need for a method, other than *naming*, to describe and identify CPs. The Copernicus problem language was introduced in the second part of this work to describe compute problems in terms of their semantics.

A. Quantifying computational problems

Prior work on algorithmic choice [6], [9] focused primarily on compute kernels, and a quantitative study of computational problems in contemporary workloads is thus of interest. A characterization of the constituent CPs in 21 real-world applications is presented. It is conjectured that, if CPs, occupying a large fraction of execution time, can be identified even in existing applications, a platform for algorithmic choice is likely of benefit to new applications.

In the set of applications, 16 CPs were identified, covering 55 % of the aggregate, analyzed execution time. Several of the CPs (various variants of sorting, searching, and DCTs) were common to up to four applications. The problem of finding the Viterbi path and the class of CPs covering matrix operations each covered more than 10 % of the analyzed execution time. The broad diversity of real-world applications employed, and the fractions of execution time of the identified CPs, give credence to the hypothesis that there is an opportunity for improving application performance by algorithmic choice.

When analyzing the type of applications best suited for algorithmic replacement, it becomes apparent that applications having single CPs per procedure and multiple procedures per CP are likely to be best suited since the CPs have clear boundaries in the code. These two categories cover roughly half of the applications analyzed.

Approximately 45 % of the analyzed execution time was not identified as easily-named CPs. The analysis of application structure showed a fourth category of applications containing difficult-to-name problems. From inspection of the source code it is apparent that problems are solved, however, it is difficult to give them a clear name. These difficult-to-name CPs may

still benefit from algorithmic replacement if a method, other than naming, is available to describe them.

B. Capturing computational problems

In the second part of the work, the Copernicus problem language is introduced, aimed to capture the semantics of compute problems. The intended use of Copernicus is to describe the problems solved in algorithms such that an algorithmic choice framework can use this description to identify sections of code solving identical problems. The language explicitly captures the domain (or input), range (or output), and relation between the domain and range of problems. Difficult-to-name CPs can be described using this language, and problem descriptions can reveal potential commonality between CPs previously obfuscated by naming.

The language is designed with the goal of algorithmic choice and programmer usability in mind. The declarative language is a superset of second-order logic and has no control-flow statements; the restricted set of operators is aimed to make it non-trivial to implement algorithms in the language. Based on the logic class, it can be proven that the language can capture at least all problems in the polynomial-time hierarchy.

To analyze the expressiveness of the language, and to make an approximation of the overhead of such semantic descriptions for algorithmic choice, the previously identified CPs were implemented in Copernicus. In total, 13 problems were successfully implemented, showing that the language is expressive enough to capture an important fraction of problems relevant to real-world applications.

The sizes of the example compiled representations in Copernicus are between 108 and 704 bytes. Given the comparison with the size of algorithmic implementations of three problems, and taking into account that the quantification showed that describing a few CPs can cover a large fraction of execution time, the additional overhead of the description for algorithmic choice is not considered an issue.

C. Future work

There are still issues to be solved before a complete framework for algorithmic choice based on the Copernicus language can be created. The Copernicus problem definition is to be compiled to an optimized format to expose as much commonality between CPs as possible. Based on this format, a framework should be able to find and match identical CPs. This raises problems since the compiled format should ideally be in a canonical form. Programmer-related artifacts, such as variable names, have to be removed, and there is the need for optimizations in the Copernicus compiler to reveal more commonality between problems. Examples of optimizations might be function inlining (problem structure is, after all, a programmer's artifact) or algebraic optimizations.

It is important to realize that the Copernicus language is not limited to algorithmic choice. The potential of problem analysis has already been mentioned in this work; examples are computational complexity and parallelism analyses. Performing analysis on the CP level can be of interest in the study of computational problems versus algorithms. The Copernicus compiler can be extended to perform these analyses.

References

[1] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, pp. 8–21, 1985.

[2] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," in *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, (New York, NY, USA), pp. 274–283, ACM, 1998.

[3] D. Bhandarkar and D. W. Clark, "Performance from architecture: Comparing a RISC and a CISC with similar hardware organization," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, (New York, NY, USA), pp. 310–319, ACM, 1991.

[4] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[6] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, (New York, NY, USA), pp. 38–49, ACM, 2009.

[7] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pp. 307–318, ACM, 1998.

[8] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge, "The limits of instruction level parallelism in spec95 applications," *SIGARCH Computer Architecture News*, vol. 27, pp. 31–34, March 1999.

[9] J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable and adaptive multi-core heterogeneous computing," in *Proceedings of the 2010 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, pp. 115–124, 2010.

[10] M. Frigo, "A fast fourier transform compiler," *SIGPLAN Notices*, vol. 39, no. 4, pp. 642–655, 1999.

[11] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–273, 2005.

[12] J. Backus, "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, pp. 613–641, 1978.

[13] M. Broy, "Declarative specification and declarative programming," in *Proceedings of the Sixth International Workshop on Software Specification and Design*, pp. 2–11, 1991.

[14] N. Immerman, "Languages which capture complexity classes," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, (New York, NY, USA), pp. 347–354, ACM, 1983.

[15] K. M. Chandy and J. Misra, *Parallel program design: a foundation*. Addison-Wesley Pub. Co., 1988.

[16] P. Stanley-Marbell, "Sal/svm: an assembly language and virtual machine for computing with non-enumerated sets," in *Proceedings of the 2010 workshop on Virtual Machines and Intermediate Languages*, VMIL '10, pp. 1:1–1:10, ACM, 2010.

[17] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," The Bell System Tech Journal, vol. 49, no. 2, AT&T, 1970.

[18] R. Jongerius, P. Stanley-Marbell, and H. Corporaal, "Quantifying the common computational problems in contemporary applications," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, 2011.

[19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, pp. 3–14, 2001.

[21] R. Jongerius, "Copernicus: Language definition for a notation of computational problems." IBM technical report, to be published, 2011.

[22] N. Immerman, *Descriptive Complexity*. Springer-Verlag, 1998.