RZ 3885 (#ZUR1411-048) 11/15/2011 (extended version: **11/24/2014**)
Computer Science 13 pages

Research Report

Quantifying the Common Computational Problems in Contemporary Applications*

Rik Jongerius**
IBM Research – Netherlands
r.jongerius@nl.ibm.com

Phillip Stanley-Marbell**
Massachusetts Institute of Technology
psm@mit.edu

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

*This work was presented at the 2011 IEEE Int'l Symp. on Workload Characterization "IISWC-2011," and a one-page summary appeared in the proceedings.

**Work performed while at IBM Research – Zurich, Rüschlikon, Switzerland

Quantifying the Common Computational Problems in Contemporary Applications

Rik Jongerius*
IBM Research — Netherlands
r.jongerius@nl.ibm.com

Phillip Stanley-Marbell*
Massachusetts Institute of Technology
psm@mit.edu

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

***Abstract*—The workloads of computing systems have traditionally been characterized in terms of the behavior of *algorithms* and their embodiments in *applications*, executing on a variety of hardware architectures. Algorithms are however only a *means* to the end goal of the solution of *computational problems*. A quantitative characterization of the constituent computational problems in contemporary applications is thus of great interest, given the recent advent of programming languages and system software platforms which enable performance and energy-efficiency tradeoffs through the exploitation of *algorithmic alternatives* for a given compute problem.**

This article quantifies the potential for the exploitation of *algorithmic choice* as a means by which applications may gain performance improvements in return for architectural or programmer investment. This argument is made through the quantitative characterization of the dominant computational problems contained in a suite of 21 applications, representative of a diverse collection of general-purpose real-world software. It is demonstrated that almost 40 % of the aggregate execution time of the applications studied is spent in a set of 16 problems that can be defined in terms of notation for describing computational problems independent of algorithms for their solution. The properties of compute problems which would make them amenable to representation in the proposed notation are discussed, and quantified through hardware performance counter measurements. Based on the insights obtained from the work presented, a tentative hardware microarchitecture for exploiting algorithmic choice is introduced.

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 earliest computing systems were achieved through the use of successively more complex instruction set architectures, with the goal of pushing more work into hardware; these architectures were ultimately limited by the difficulty of compiling general-purpose applications to target complex hardware [1], [2], [3]

As an alternative to complex instruction set computing (CISC) platforms, reduced instruction set computing (RISC) systems eschewed the hardware implementation complexity and instruction non-uniformity of CISC architectures for a strategy of simplified hardware, easier targeting of compilers, and the possibility of higher clock frequencies facilitated by the reduced cycle time of (pipelined) simplified hardware [1]. Coupled with semiconductor process technology scaling and improvements in clock frequencies, the RISC revolution pushed performance further through the following two decades. In the last few years however, due to the inability to cope with the increases in power dissipation and challenges of heat removal associated with ever-higher clock frequencies, there has been a move towards obtaining performance through parallelism [4]. During each of the aforementioned phases of evolution of computing systems, the respective performance growth vehicles were viewed as the “one true” solution, only to be replaced, often disruptively, by a different approach when the erstwhile growth vehicle could no longer scale.

Performance scaling through parallelism, the current growth vehicle, also has its limits. Scaling through parallelism is typically classified as *strong scaling*, if performance gains are achieved when problem sizes remain fixed as hardware concurrency is increased. In contrast, the term *weak scaling* is used to refer to the situation when increases in hardware concurrency alone might yield little performance gains, but where data-level parallelism can be exploited in making problem sizes larger as hardware concurrency is increased. Strong scaling will ultimately be limited by Amdahl’s law [5] and by the amount of parallelism available in application code, while weak scaling will be constrained by the data-level parallelism available in applications paired with input datasets.

All of the aforementioned performance vehicles—complex hardware or hardware acceleration, clock speed growth, and parallelism—have the same fundamental goal: making (fixed) *algorithms* execute faster. In principle, however, computing devices are intended to solve *computational problems*, which are conceptually separate from specific algorithms or implementations thereof in applications. Thus, in principle, if the semantics of computational problems can be exposed to

*Work performed while at IBM Research—Zürich, Rüschlikon, Switzerland

hardware, to the operating system, or to language runtimes, *algorithmic choice* could be made a vehicle for performance growth. For example, for the computational problem of sorting a collection of items in lexicographic order, the algorithm or implementation thereof that achieves best performance (or, say, energy-efficiency) will depend on the type of elements, the number of elements, as well as the target execution architecture. If the computational problem could be specified in a machine-readable form, in much the same way computational algorithms are specified in (assembly) machine language, a choice of algorithms (or the development of new algorithms) could be used as a new growth vehicle for performance beyond parallelism.

This work addresses two important questions pertaining to the potential of algorithmic choice for computational problems to serve as a future vehicle for performance growth. First, the types of applications which lend themselves to separation of their constituent problem definitions from algorithms for their solution, are studied. Second, the feasibility of identifying well-defined computational problems, occupying a significant portion of execution time in existing complex software applications, is studied. It is conjectured that, if such well-defined components can be identified even in *existing* software, and if these components are amenable to being described independent of algorithms for their solution, then providing *problem description* facilities to implementers of *new* software is likely to be of benefit. Problem descriptions added as annotation to source code or inserted into existing binaries could then be used to facilitate hardware- or system-software-driven algorithmic choice.

Following the discussion of relevant related work in Section II, Section III introduces the notion of computational problems independent of specific algorithmic implementations, by means of an example. The methodology used to identify such algorithm-independent computational problems in applications is introduced in Section IV, along with a set of 21 applications which are used to motivate the study of algorithmic choice as a performance growth vehicle. Section V presents and discusses the results of the identification of the dominant algorithm-independent compute problems in the corpus of applications. Section VI summarizes the paper and discusses a tentative architecture for exploiting algorithmic choice as a performance growth vehicle.

II. Related Work

Workload characterization has traditionally involved analyzing the properties of compiled applications, paired with their input datasets, executing on real hardware platforms such as shared-memory multiprocessors [6] or simulated hardware platforms such as microarchitectural simulators [7]. To quantify the potential of algorithmic choice as a performance growth vehicle, however, what is desired is an understanding of the constituent compute problems (semantics) that occur in contemporary applications.

Recognizing the potential for great differences in performance from various implementations of the same algorithm, the Berkeley computational motifs [8] are an attempt at a

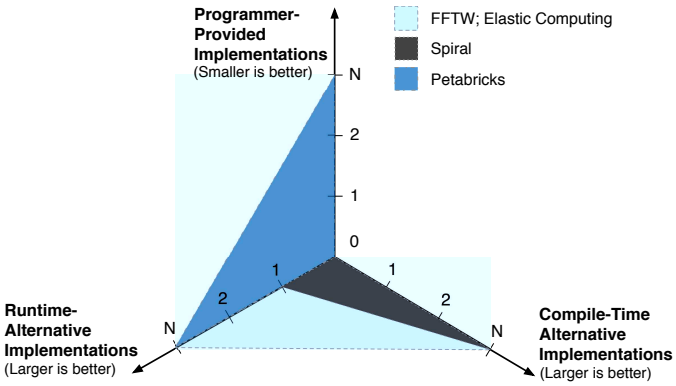


Figure 1. Summary of commonalities between four languages / runtime systems to which the workload characterization presented in this work is relevant—PetaBricks, FFTW, SPIRAL, and Elastic Computing.

classification of applications based on the constituent numerical methods of computational problems. *Design patterns* [9], a refinement of the ideas of the Berkeley computational motifs, are a proposed framework for software architecture, comprising five categories of patterns which can be composed to describe the architecture of an application. Both design patterns and the Berkeley motifs move away from classifying applications based on specific implementations, but however provide neither a quantitative study of the constituent (or common) computational problems occurring in applications, nor a concrete description of the semantics of the problems whose execution dominate runtime.

One recent attempt to put the dwarf classification on a more quantitative footing analyzes instruction-level parallelism (ILP) and thread-level parallelism (TLP) of applications which, based on their dwarf classification, belong together [10]. This approach, however, treats each application as a black box, without quantifying what *computational problems* dominate the execution time for which ILP and TLP are reported.

Having a quantitative analysis of the computational problems that are solved in a specific application enables reasoning about the potential for the use of algorithmic replacement. However, to enable a platform to benefit from algorithmic choice requires a supporting framework. Recent examples of such frameworks include PetaBricks [11], 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 vein, the elastic computing framework [12], provides a library containing multiple algorithms per computational problem, permitting a programmer to call a function (by a statically-defined name binding) to solve the statically-named problem. The characterization presented in this article provides the first quantitative analysis of the potential applicability of frameworks such as these.

In contrast to the aforementioned general-purpose programmer-driven frameworks, FFTW [13] is a domain-

specific framework which tunes fast Fourier transform (FFT) algorithms to a target platform. Similarly, the SPIRAL [14] framework permits a programmer to specify a DSP linear signal transform which is then optimized and converted into code targeted to a given platform. Both FFTW and SPIRAL are, however, domain-specific.

Figure 1 illustrates the differences in the work performed by a programmer, as opposed to the compiler or runtime system, across the aforementioned frameworks. From an application programmer's perspective, it is considered worse if more implementations have to be programmed as this incorporates more work. For algorithmic choice, however, having more more implementations available at compile- or run-time results in a better tuning of problem implementation to the platform.

When dedicated ASICs are employed in solving computational problems, they often result in a significant improvement in compute efficiency in comparison to general-purpose processors [15], or in an improvement of the energy-efficiency under power usage limits [16]. The characterization of the occurring computational problems in contemporary applications presented in this article provides a much needed quantitative basis for the potential gains that can be achieved by all the aforementioned compiler, runtime, and hardware techniques.

III. Computational Problems

In the remainder of this article, the term *computational problem (CP)* will be used to refer to the semantic properties of a computation, which determine what relation exists between the inputs and outputs of the computation, as opposed to *how* that desired relation is achieved.

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, which are 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 (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. For example, for the compute problem of sorting a list of strings, the input set is a set of strings; the output is a set of tuples of integers and strings, the integers corresponding to the ordinal position of the strings when lexicographically sorted.

In principle, a CP could be defined at any level of granularity, from very fine-grained (e.g., the CP corresponding to the sum of two integers), to very coarse-grained (e.g., the CP corresponding to a whole application). In practice, CP definitions are most meaningful when they correspond to a non-trivial problem that can be expressed in the context of Definition 1, in less space than it would take to give an explicit algorithmic implementation. As will be quantified in Section V, the execution times of many real-world applications

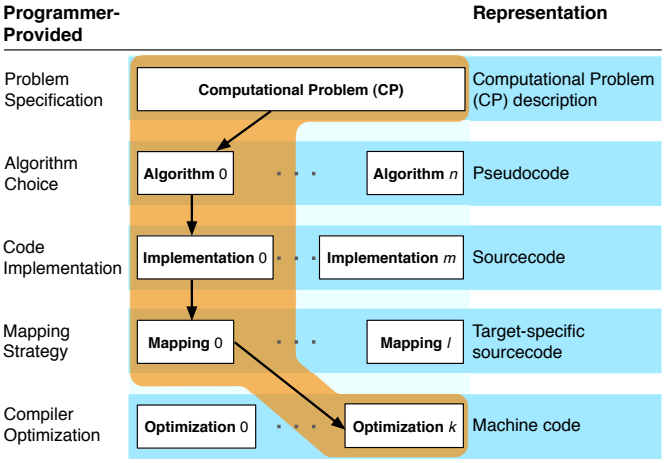


Figure 2. Computational problems: a CP can be solved with one or more different algorithms, and for each algorithm, different implementations can be created. Each implementation can then be mapped to different platforms and compiled using different compiler optimizations.

are dominated by compute problems that have this property. There are however also certain types of applications, or sections thereof, that are a poor fit for description as CPs; the characteristics of these outlier applications are quantified in Section V.

A. Computational problems versus algorithms

The collection of CPs that comprise an application determine *what* the application does. However, the CPs themselves do not imply the use of a specific algorithm, as the relation between the inputs and outputs could be achieved by a variety of algorithms (e.g., quicksort versus mergesort for the sorting CP).

During the course of implementation of an application, a programmer makes a variety of architectural choices before the final binary is available for execution, as illustrated in Figure 2. Multiple algorithms may exist for solving a given CP; in implementing a given algorithm, there may likewise 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 2, do not inherently affect the semantics of an application, but however often have significant effects on performance or energy-efficiency. Understanding the constituent CPs in applications, and even possibly having a means of capturing these CP definitions in the same way that binaries capture a codification of algorithms, 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 this article shows the potential opportunities for implementation of such techniques.

Table I
DESCRIPTION OF THE k-MEANS CP IN TERMS OF THE COMPONENTS GIVEN IN DEFINITION 1—*domain*, *range*, AND *relation*.

Domain	A set of points, $\mathbf{P}$, and required number of clusters, k.
Range	A clustering into k sets $\mathbf{S} = \{S_1, S_2, ..., S_k\}$.
Relation	$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{p}_j \in S_i} ||\mathbf{p}_j - \mu_i||^2$, where μ_i is the mean of points in S_i.

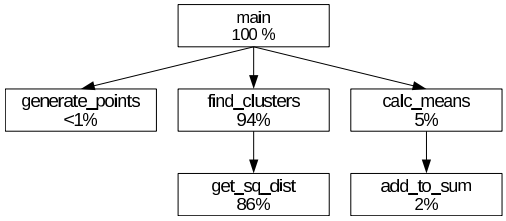


Figure 3. Call graph of the k-means clustering application. Only procedures relevant to the k-means clustering algorithm are shown. The numbers show the percentage of execution time spent in a procedure and its children. For example, the `main()` procedure accounts for 100 % of the run time, of which 94 % of the time is spent in `find_clusters()`, 5 % in `calc_means()` and less then 1 % in `generate_points()`.

B. Example: k-means clustering

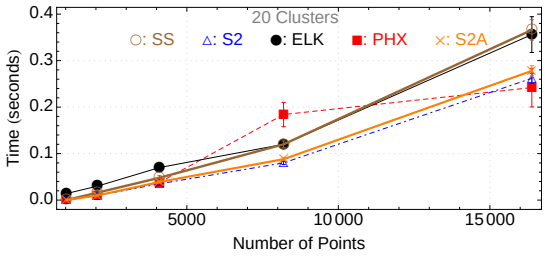
The computational problem of k-means clustering is used in what follows to illustrate the concepts of CPs in the context of a real problem, algorithms for solving the problem, and different implementations of these algorithms. The concept behind k-means clustering is straightforward: given a set $\mathbf{P}$ of points in an n-dimensional space, the problem is to group the points into k groups such that the square error of distances from points in any cluster to the centroid of the cluster is minimized. The description of the k-means CP in terms of the components of Definition 1 is shown in Table I. For any given set of input points and output result, the only restriction on a valid k-means algorithm is that it satisfies the CP definition given in the table.

Several different algorithms exist for solving the k-means CP, including algorithms by MacQueen [17], Elkan [18], Hartigan and Wong [19], as well as algorithms based on ideas from principle component analysis [20]. Figure 3 shows an abbreviated procedure call graph obtained from gprof [21] profiling of one implementation of the MacQueen k-means algorithm. The implementation in question is the serial k-means clustering implementation from the test applications of Phoenix MapReduce runtime system distribution [22]. From Figure 3, the computation in that implementation is dominated by the routine `get_sq_dist()`, which computes the Euclidean distance between two points (i.e., the 2-norm). The CP definition for the Euclidean distance calculation is given in Table II, and provides an example of a sub-computational problem which will likely occur in other applications. The quantitative survey presented in Section V gives insight into what coarse- and fine-grained CPs occur in contemporary applications.

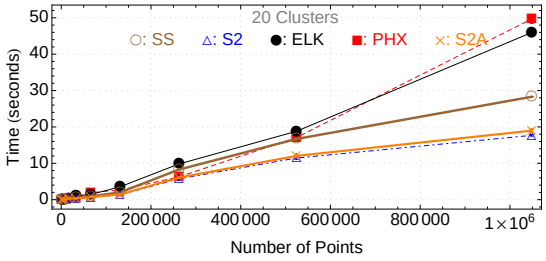
The potential opportunities for taking advantage of different algorithms and implementations thereof, is illustrated for the k-means CP in Figure 4. The figure plots the execution time of five different k-means implementations, across a range of input set sizes (2^{10} to 2^{20}), and as a function of the number of target clusters (20 and 200 clusters). The measurements were taken on a dual-core 2.53 GHz Intel Core i5 system, running MacOS 10.6.7. Two of the implementations (PHX and SS) are alternative serial implementations of the same (MacQueen) algorithm. One implementation (ELK) implements the algorithm by Elkan, which attempts to reduce the number of 2-norm computations, previously shown to dominate compu-

Table II
DESCRIPTION OF THE EUCLIDEAN DISTANCE SUB-CP FROM k-MEANS CLUSTERING.

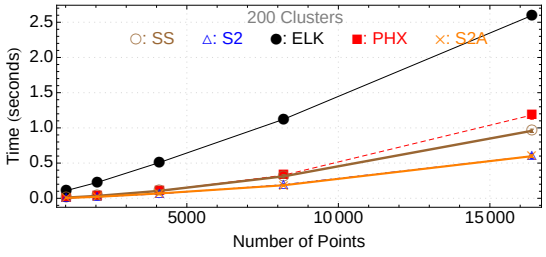
Domain	Two n-dimensional points, p and q.
Range	A distance, d.
Relation	$d = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$.



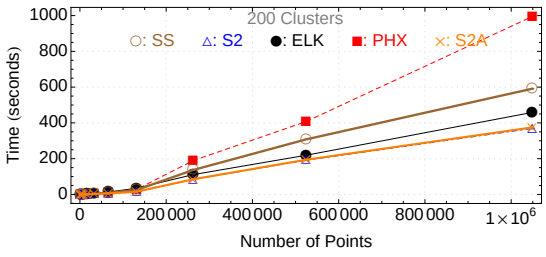
(a) k-means variants, 20 target clusters, 2^{10} to 2^{14} points.



(b) k-means variants, 20 target clusters, 2^{10} to 2^{20} points.



(c) k-means variants, 200 target clusters, 2^{10} to 2^{14} points.



(d) k-means variants, 200 target clusters, 2^{10} to 2^{20} points.

Figure 4. Comparison of performance of single threaded MacQueen k-means algorithm in PHX, and SS, the parallel versions in S2A, and S2 and the single threaded Elkan algorithm ELK. The execution time is plotted for up to 16 k points for both $k = 20$ clusters (a) and $k = 200$ clusters (c). To show the general trend, for both cases the results are also plotted up to 1 M points. The error bars in the plots show the standard deviation across 5 independent runs.

tation time in the call graph of Figure 3. Lastly, two parallel OpenMP implementations of the MacQueen algorithm (S2A and S2) were evaluated, parallelized over two threads; the only difference between these two parallel implementations was the manner in which threads were synchronized. All five implementations were run with the same input sets, and the analyses were replicated to enable calculation of the variances across timing runs.

From Figure 4, there is no single algorithm, or implementation thereof, which is *always* fastest. Comparing the three different single-threaded implementations (PHX, SS, and ELK) for $k = 20$ clusters, SS is significantly faster for 2^{20} input points. For smaller input sizes, except for 2^{13} input points, PHX is faster. For $k = 200$ clusters, for input sizes larger then 2^{18} ELK is the fastest single threaded algorithm, for smaller input sizes PHX and SS have similar performance, SS being slightly faster.

The parallel implementations S2A and S2 use two threads, and for inputs with greater than 2^{17} points, they clearly outperform all the serial implementations. For small data sets however, the run times are in some cases worse than a single-threaded implementation. For example, PHX outperforms both S2A and S2 for $k = 20$ clusters and 2^{14} points.

To extend analyses such as the above for the k-means clustering problem to large real-world applications, it is essential to identify which CPs dominate execution time and to determine whether they can be expressed independent of their implemented algorithms; such an ability to be described as a CP is an indicator of the potential availability of alternative algorithms that solve the same problem, as well as, naturally, alternative implementations thereof. Section IV, which follows, presents the methodology used in the remainder of this article in performing such quantitative characterization.

IV. Methodology

To quantify the presence of well-defined CPs in real-world applications that can be expressed independent of algorithms for their solution, a set of applications, from a diverse range of domains, was studied.

A. Application suite

The applications used to study the occurrence of well-defined algorithm-independent CPs in contemporary applications were taken from the SPEC CPU2006 [23] and MiBench [24] benchmark suites. The rationale for using well-known benchmark suites is that these suites have already been pre-selected as being representative of contemporary workloads in both the desktop / server market (SPEC) and the embedded / low-power computing market (MiBench). Since some of the applications in the MiBench suite are better regarded as *kernels* rather than full-fledged applications, only the larger applications from the MiBench suite were employed. The 21 applications employed, from the aforementioned two suites, are listed in Table III, along with indications of their general application domain and dominant type of arithmetic (e.g., integer versus floating-point arithmetic).

Table III
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

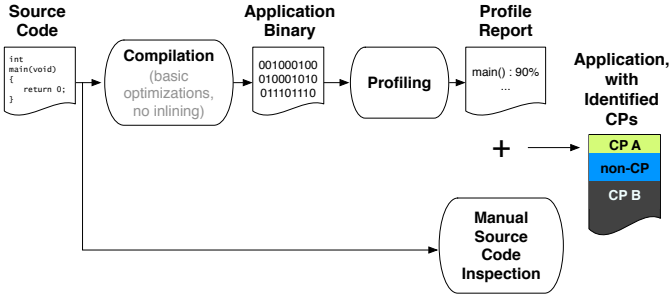


Figure 5. Methodology used to identify computational problems in applications. The applications were compiled and profiled, and the results from profiling together with manual code inspection were used to identify the CPs.

B. CP identification

The quantification of the constituent CPs in applications ultimately requires a manual analysis and understanding of the source code of the applications involved. Since the applications employed in the study comprise, in some cases, hundreds of thousands of lines of code, a three-pronged approach was employed. First, the applications were profiled using gprof, to identify the top five subroutines in the execution time breakdown. In some applications however, a majority of the application was contained in a single function, reducing the utility of the identification of the top five functions in the previous gprof analysis step. In these cases, the applications were profiled again, in a second step, using the OProfile [25] hardware-based profiling facility, to obtain execution time percentage breakdowns at the individual-program-statement and loop-nest level. The subroutines identified in the first two stages were then analyzed, in a third stage, by studying the source code. From code inspection, a description of the CP being solved, in the form described in Section III, was

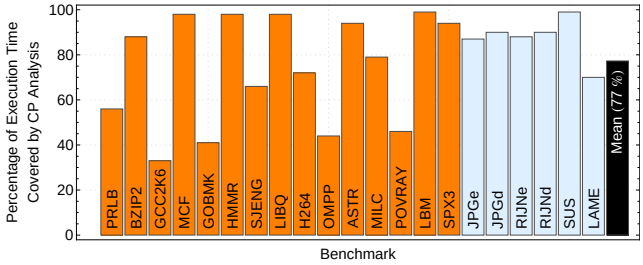


Figure 6. Percentage of execution time analyzed per application on Atom D510. For example, the source code analyzed for PRLB accounts for 56 % of the execution time of that application.

obtained, if possible. Figure 5 illustrates the overall flow of the CP identification methodology employed in the remainder of this article.

V. Quantitative Characterization of CPs

The methodology described in Section IV was applied to the applications listed previously in Table III. For the profiling steps, the applications were compiled with optimization flags `-O1 -g`; higher levels of optimization were not employed, to ensure that the obtained gprof and OProfile profiles were in close correspondence with the structure of the original source code, and that subroutines were not inlined.

The second phase of the characterization (statement-level analysis) was performed on a 1.6 GHz Intel Atom D510, running Linux kernel 2.6.35, with the OProfile hardware performance counter kernel module.

In the third phase of profiling, the top five subroutines and loop nests, identified in the previous two phases as dominating execution time, were analyzed by manual source code inspection. For each of the applications analyzed, Figure 6 lists the percentage of execution time covered by all three phases of the analysis. Thus, for example, for the benchmark 400.perlbench, the region of source code identified for detailed analysis in the third and final phase occupied 56% of the total execution time of the application. For several applications (e.g., 470.lbm and Susan) up to 99 % of the execution time was covered by the three analysis phases. The smallest fraction of execution time covered occurred in the case of 403.gcc, where the analyzed source code only accounted for 33 % of the total execution time. In general, when a large number of subroutines contributed to the total execution time, the fraction of total execution time consumed by the top five subroutines was lower, and the coverage percentage in such situations was therefore also lower. In all, the fraction of source code analyzed for the complete set of applications covers 77 % of the aggregate execution time of the entire suite of applications.

The degree of difficulty of manual inspection of source code to identify CPs depends on the structuring of the application in question, and on 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; in a few cases, however, 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 IV. The amount of effort required to specify the CPs implemented in an application, independent of the algorithms for their solution, varies across the categories in Table IV. If these identified CPs are also intended to be candidates for algorithmic replacement, it is desirable to have well-defined boundaries within a program (e.g., procedures) at which such replacement may take place; this is also captured in the construction of the four categories. Approximately half of the applications fall in the two categories (categories 1 and 3) which have such well-defined CP boundaries. For applications where multiple CPs occur within a single procedure (category 2), it may still be straightforward to divide such a single procedure into multiple procedures, if necessary.

A. 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 can easily be identified as a well-defined CP. One example of an application in this category is 433.milc. The top five procedures which dominate execution time, as well as most leaves of the call graph, implement an isolated CP, in this case a CP related to matrix arithmetic. There is, for example, a procedure which performs matrix-matrix multiplication, while another procedure performs matrix addition.

Applications in this category are typically implemented using a clear structure, and the algorithms have clear entry and exit points—the procedure call and return.

B. 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; in what follows, this additional 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 procedure. For example, in two of the identified procedures in 464.h264ref, three CPs 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.

C. 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 well-defined CPs. However, when looking at multiple procedures, it becomes apparent that, together, they solve a *minimum-cost network flow* problem (using a *network simplex* algorithm).

Similar to category 1, applications in category 3 have CPs that have well-defined boundaries in the application's structure, but in this case with multiple procedures per CP. Thus, a single procedure can still serve as the entry and exit point of a CP.

Table IV
CATEGORIZATION OF APPLICATIONS BASED ON THE PROGRAM STRUCTURE AND TYPE OF COMPUTATIONAL PROBLEMS SOLVED. CATEGORIES 2 AND 3 BOTH CONTAIN 401.BZIP2, AS 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** (Algorithm-Specific or Control-Dominated CPs: *Little or No Algorithmic Choice*)
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

D. CP category 4: algorithm-specific or control-dominated

There are multiple applications which have procedures for which it is difficult to identify the problem being solved (in terms of Definition 1), but which are also not part of a larger, well-defined CP; such applications account for about a third of the applications studied. Mostly, these procedures are program-specific and have an irregular control-dominated structure. For these procedures, it is expected that there is not much (if any) algorithmic choice. Applications such as 458.sjeng fit in this category: its procedures are control-dominated and specific to the semantics of a chess-playing engine.

E. Insights from categorization

Figure 7 shows the instruction mix (branches, loads, stores, floating-point, and other instruction types) in the applications studied. It is observed that applications falling in the first two categories of Table IV 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 manual source code inspection. As the CPs identified in category 3 are large and more complex, they were, as a result, naturally split across multiple procedures; it is thus also not surprising that they have more branch / control instructions.

There are, however, two exceptions. The JPEG encoder and decoder (JPGe and JPGd), from Figure 7, have above-average fractions of control instructions, but are classified in category 1. The higher fraction of control instructions in this case is expected to be 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 because the application is essentially defined by the algorithm that it implements. It is thus meaningless to define a CP for Rijndael encode or decode independent of the Rijndael algorithm, unless, e.g., the CP definition is instead defined in terms of a data encryption problem, with constraints on, e.g., some measure of cryptographic strength.

Overall, approximately two-thirds of the applications studied fall into categories 1–3, and could potentially benefit from alternative algorithms for solving the identified CPs. Of particular importance, based on the information in Table IV, such CPs can be relatively easily identified, as they are embodied either in well-defined code regions such as subroutines, or collections thereof. Section V-F, which follows, details the specific CPs identified, and provides quantitative analysis of the fraction of the total and per-application run-times dominated by the identified CPs. As will be demonstrated, the identified CPs occupy a large enough fraction of execution time to make it worthwhile to find alternative algorithms for their solution.

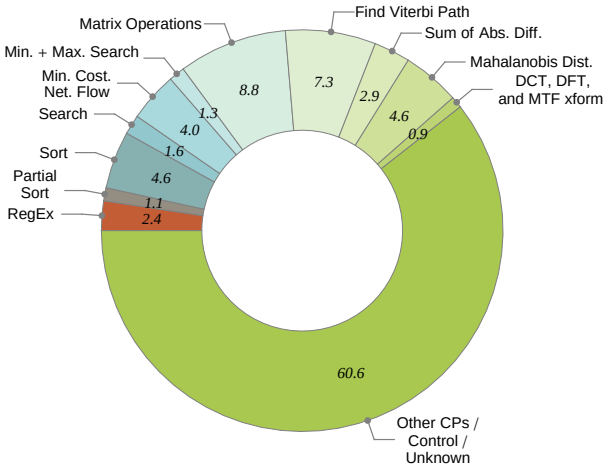


Figure 8. Percentages of total execution time of the complete set of applications covered by the identified computational problems.

F. Identified CPs and their frequencies of occurrence

Table V 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 V, all CPs related to matrix arithmetic are grouped under a "Matrix Operations" CP for brevity of exposition, even though, e.g., matrix multiplication, matrix addition, etc., are indeed distinct CPs.

Figures 8 and 9 show the fraction of execution time covered by the identified CPs in the complete set of applications, and the MiBench subset, respectively. The DCT, DFT, and move-to-front transform are condensed in one slice for clarity of the figure. In total, the small set of 16 identified CPs covers approximately 39 % of total execution time of all applications.

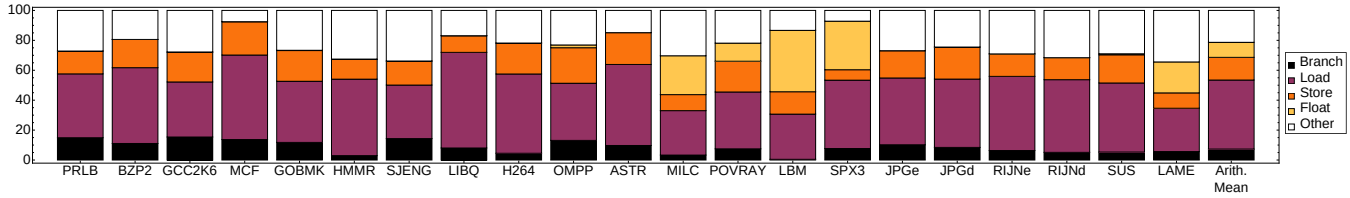


Figure 7. Instruction mix breakdown. Applications from category 4 in Table IV generally also have high fractions of control instructions.

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

Computational Problem	Application	Execution Time (s)	% of Application
RegEx	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

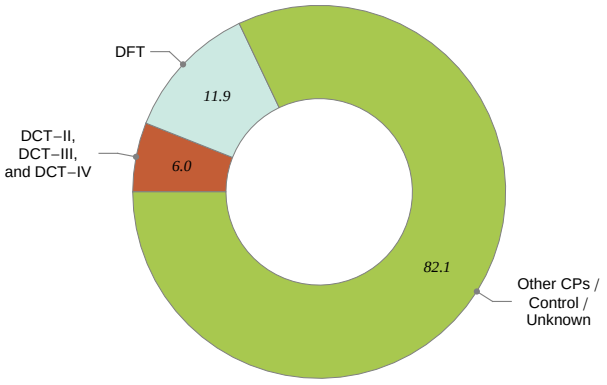


Figure 9. Percentages of total execution time of the MiBench subset of applications covered by the identified computational problems.

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

Figure 10. Per application breakdown of the execution time of identified CPs.

of execution time, while (partial) sorting accounts for at least 5.6 % of the total runtime, making (partial) sorting one of the most important CPs.

In terms of execution time, CPs related to matrix arithmetic and the computational problem of finding the Viterbi path account for the largest fractions, at 8.3 % and 7.3 %, of the aggregate execution time of the set of applications. Both problems, however, appear only in two applications each and are responsible for a large amount of run time in only a restricted set of applications.

Regular expression matching, the move-to-front transform, finding the minimum-cost network flow, the sum of absolute differences, and the Mahalanobis distance only occur in single applications. While these CPs might seem to be of less interest, they, individually, can still account for up to 4 % of the total execution time of applications. However, we can still envision them to be common to many applications. A straightforward example is regular expressions 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.

Figure 10 shows the breakdown of identified CPs in four of the applications with respect to execution time of just those applications. Across applications, the fraction of time for which CPs were identified is different. For example, for 429.mcf about 88 % of the execution time is covered by the

identified CPs. On the other hand, for lame, only 31 % is covered. The other two examples, 464.h264ref and 482.sphinx3, are in between, with 44 % and 49 % of the execution time covered by the identified CPs.

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 the CPs, can potentially gain more than applications with a low fraction. The unidentified fractions of execution time in Figure 10, however, are not necessarily non-candidates for algorithmic choice: there may still be more potential CPs to be identified in the parts not covered by manual code inspection. However, based on the insights gained from the categorization, some applications have large amounts of glue logic (I/O, setting up data structures, etc.) or may have highly control-dependent code, precluding their description in terms of CPs.

Although there are thus obviously portions of the applications investigated that will not gain from algorithmic choice, at least 39 % of the total execution time of the applications is covered by 16 well-defined CPs (which can be described in the context of Definition 1). 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 hardware architectures. Table VI gives the domain and range in the context of Definition 1, for each of the 16 identified CPs. The domains and ranges given in the table are specified informally, and not in terms of actual sets (compared to, e.g., Table I). Their formalization, as well as the complete specification of the third component of the CP—the relation—are items of ongoing work.

Table VI
DOMAIN AND RANGE FOR THE CPS IDENTIFIED IN THE SET OF APPLICATIONS.

RegEx	
Domain	A string s A regular expression in the string r
Range	A modified string s A Boolean b indicating a match
Sorting and Partial Sorting	
Domain	A set of elements S
Range	A sorted set of elements T
Move-To-Front Transform	
Domain	A string s A string with the alphabet t
Range	A sequence of integers M
Search	
Domain	A set of elements S An element t
Range	An index i into the set S
Maximizing/Minimizing Search	
Domain	A set of elements S
Range	An index i into the set S
Minimum-Cost Network Flow	
Domain	A flow network $G\,(V, E)$ A source $s \in V$ A sink $t \in V$ A list of capacities $c\,(u, v)$ A list of costs $a\,(u, v)$
Range	A list of flows $f\,(u, v)$
Matrix Operations	
Domain	A matrix M A matrix N
Range	A matrix O
Finding Viterbi Path	
Domain	A hidden Markov model H A sequence of observed outputs X
Range	A sequence of states Y
Sum of Absolute Differences	
Domain	An array of integers D
Range	An integer s
DCT-II, DCT-III, DCT-IV, DFT	
Domain	A set of time-domain samples x
Range	A set of frequency-domain samples X
Mahalanobis Distance	
Domain	An n-dimensional vector x An n-dimensional vector μ A covariance vector v
Range	A distance d

VI. SUMMARY, DISCUSSION, AND FUTURE DIRECTIONS

This work presented a quantitative characterization of the constituent *computational problems (CPs)* in 21 real-world applications. CPs capture the *semantics* of computations (i.e., the *problem* solved), independent of specific *algorithms* for implementing those computations. The quantitative characterization of the occurring CPs in contemporary applications provides insight into the potential for *algorithmic choice*—the substitution of one algorithm solving a given CP, by another solving the same CP. The exploitation of algorithmic choice is a potential path for gaining performance improvements in future computing systems, in the same manner that clock frequency improvements from technology scaling provided, at the height of the RISC era, and in the manner in which core count is today the dominant means of computing system performance gains. Despite the recent interest in algorithmic choice as a means for improving performance [11], [12], there has hitherto been no quantitative study of contemporary workloads, to identify the occurring compute problems and candidates for algorithmic replacement; this article remedies this deficiency.

It is conjectured that, if CPs can be identified even in existing legacy applications, then it will be reasonable to expect implementers of new applications to be able to annotate their applications with CP definitions. Such annotations, if embedded in application binaries, could be used by future systems to facilitate performance improvements of applications, without the need for re-compilation. A language such as the notation for computational problems (NCP) [26] can be used for such purposes. The Appendix describes several of the identified CPs in this work in NCP.

Applications having single CPs per procedure and multiple procedures per CP, are likely to be best suited for algorithmic replacement, as the CPs have clear entry and exit points in the code. These two categories cover approximately half of the applications analyzed in this article; on the other hand, applications having multiple CPs per procedure may require restructuring of the program to isolate the CPs. Algorithm-

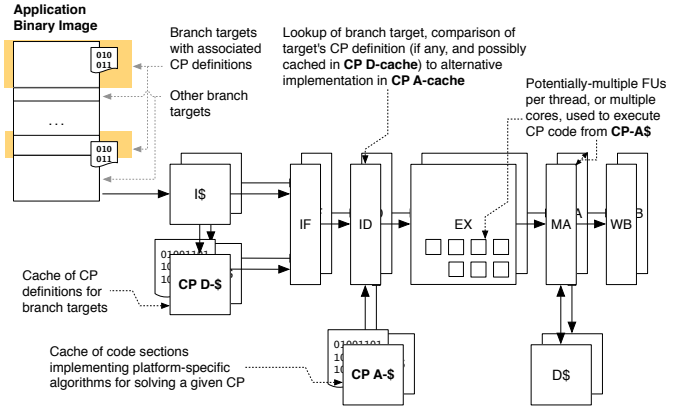


Figure 11. An example of a standard processor pipeline augmented with cache structures to aid acceleration via algorithmic choice.

specific CPs (such as Rijndael encode and decode) and control-dominated CPs (such as large portions of the Gcc compiler) constitute approximately 33 % of the execution time of the applications studied, and are unlikely to be suited to algorithmic replacement. It was observed that, in general, applications in which one or more well-defined CPs could be identified were not control-dominated.

In the set of applications analyzed, 16 well-defined CPs were identified, covering almost 40 % of the aggregate execution time of the complete set. Several of the CPs identified (various variants of sorting, searching, and DCTs) were common to up to four different applications. Of the identified CPs, one single class of CPs, matrix operations, accounted for more than 8 % of the aggregate execution time. The aforementioned statistics are naturally dependent on the applications in question; however, the broad diversity of real-world applications employed, and the sizes of the execution times of identified CPs, gives credence to the hypothesis that there is an opportunity for improving application performance by algorithmic substitution. One direction of our ongoing efforts is to quantify the actual speedups that can be achieved, by identifying multiple alternative algorithms for each CP in categories 1 through 3 of Table IV, as well as multiple alternative implementations, as illustrated for the k-means example in Section III, and to evaluate these on a diverse set of hardware platforms.

The architecture of one potential hardware direction is illustrated in Figure 11. The system associates branch targets with CP definitions, which must be defined in a machine-readable format. During execution of the normal instruction stream, if a branch target for which there is a CP definition in the CP-definition cache (**CP D-$**), and for which one (or more) alternative implementations exist in the CP-implementation cache (**CP A-$**), one such alternative implementation may be executed. This replacement may occur transparent to the application, yielding the same semantic result as the original unmodified application. Key to the functioning of such a system is a machine representation for CPs; we are currently investigating alternative representations, including building upon existing work in the research literature.

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, 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] 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, (New York, NY, USA), pp. 307–318, ACM, 1998.

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

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," Technical Report No. UCS/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[9] K. Keutzer and T. Mattson, "A design pattern language for engineering (parallel) software," *Intel Technology Journal*, vol. 13, no. 4, pp. 6–18, 2009.

[10] V. Caparrós Cabezas and P. Stanley-Marbell, "Quantitative analysis of parallelism and data movement properties across the berkeley computational motifs," in *Proceedings of the 8th ACM international conference on Computing Frontiers*, 2011.

[11] 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.

[12] 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.

[13] M. Frigo, "A fast fourier transform compiler," *SIGPLAN Notices*, vol. 39, no. 4, 1999.

[14] 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.

[15] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th ACM International Symposium on Computer Architecture*, 2010.

[16] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 205–218, 2010.

[17] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Math, Statistics and Probability*, 1967.

[18] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th International Conference on Machine Learning*, pp. 147–153, 2003.

[19] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 28, no. 1, 1979.

[20] C. Ding and X. He, "K-means clustering via principal component analysis," in *Proceedings of the 21st International Conference on Machine Learning*, 2004.

[21] S. L. Graham, P. B. Kessler, and M. K. McKusick, “gprof: a call graph execution profiler,” in *Proceedings of the 1982 SIGPLAN Symposium on Compiler construction*, 1982.
[22] C. Ranger, A. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating mapreduce for multi-core and multiprocessor systems,” in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.
[23] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *Computer Architecture News*, vol. 34, no. 4, 2006.
[24] 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*, 2001.
[25] W. E. Cohen, “Tuning programs with oprofile,” *Wide Open Magazine*, 2004. Premier issue.
[26] R. Jongerius and P. Stanley-Marbell, “Language definition for a notation of computational problems,” rz 3828, IBM Research, 2012.

Appendix
Identified CPs described in NCP

A notation for computational problems (NCP) is introduced by Jongerius et al. [26] and can be used to describe computational problems in terms of their input, output, and the relation between these two. Most of the CPs identified in this work we described in the NCP language and are listed here.

Most descriptions try to match the implementation of the algorithm in the benchmarks as closely as possible. However, not all side-effects of an actual implementation—which might have no relation to the algorithm itself—are captured.

Listing 1. Matrix multiplication from 433.milc or 40.lbm.

```
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]) }
 } } ;
```

Listing 2. DCT-II CP from, for example, JPEG encode.

```
#define PI 3.141592
fun cos(x : real<32>) : real<32>

domain ::
 x : real<32>[8,8]  //time domain samples
range ::
 X : real<32>[8,8]  //freq domain samples
relation ::
 Y : real<32>[8,8]  // intermediate transform
 k,l : int<32> = <0 to 7>
 n : int<32> = <0 to 7>

 exists Y,X {
  forall l, k {
   Y[l,k] == sum n { (x[l,n]
    * cos(PI / 8 * (n + 0.5) * k)) }
  } and
  forall l, k {
   X[l,k] == sum n { (Y[n,k]
    * cos(PI / 8 * (n + 0.5) * l)) }
  }
 } ;
```

Listing 3. DCT-III CP as it can be found in, for example, 464.h264ref.

```
#define PI 3.141592
fun cos(x : real<32>) : real<32>

domain ::
 N : int<32>        //# samples
 x : real<32>[N,N]  //time domain samples
range ::
 X : real<32>[N,N]  //frequency domain samples
relation ::
 Y : real<32>[N,N]  //intermediate transform
 k,l : int<32> = <0 to N-1>
 n : int<32> = <1 to N-1>

 exists Y,X {
  forall l, k {
   Y[l,k] == 0.5 * x[l,0]
    + sum n { (x[l,n] * cos(PI / N * (k + 0.5) * n)) }
  } and
  forall l, k {
   X[l,k] == 0.5 * Y[0,k]
    + sum n { (Y[n,k] * cos(PI / N * (l + 0.5) * n)) }
  }
 } ;
```

Listing 4. DCT-IV CP as it appears in Lame.

```
#define PI 3.141592
fun cos(x : real<32>) : real<32>

domain ::
 N : int<32>
 x : real<32>[N]  //time domain samples
range ::
 X : real<32>[N]  //frequency domain samples
relation ::
 k : int<32> = <0 to N-1>
 n : int<32> = <0 to N-1>

 exists X {
  forall k {
   X[k] == sum n { (x[n] * cos((PI / N)
    * (n + 0.5) * (k + 0.5))) }
  }
 } ;
```

Listing 5. DFT CP from Lame or 482.sphinx3.

```
#define PI 3.141592
fun cos(x : real<32>) : real<32>
fun sin(x : real<32>) : real<32>

typedef ::
 complex : struct {
  r : real<32>
  i : real<32>
 }
domain ::
 N : int<32>
 x : complex[N]  //time domain samples
range ::
 X : complex[N]   //frequency domain samples
relation ::
 k : int<32> = <0 to N-1>
 n : int<32> = <0 to N-1>
 exp_res : complex[N,N]

 fun real_complex_mul(a,b : complex) : real<32> ::=
  a.r * b.r - a.i * b.i ;
 fun imag_complex_mul(a,b : complex) : real<32> ::=
  a.r * b.i + a.i * b.r ;

 exists X, exp_res {
  forall k, n {
   exp_res[k, n].r == cos(-2 * PI * n * k / N) and
   exp_res[k, n].i == sin(-2 * PI * n * k / N)
  } and
  forall k {
   X[k].r == sum n {
    real_complex_mul(x[n], exp_res[k, n])
   } and
   X[k].i == sum n {
    imag_complex_mul(x[n], exp_res[k, n])
   }
  }
 } ;
```

Listing 6. CP to calculate the Mahalanobis distance as found in 482.sphinx3.

```
fun sqrti(x : int<32>) : int<32>

domain ::
 N : int<32>
 x : int<32>[N]     //multivariate vector
 mu : int<32>[N]    //means
 S : int<32>[N,N]   //covariance matrix
range ::
 d : int<32>        //distance
relation ::
 n,m : int<32> = <0 to N-1>
 T : int<32>[N]     //temporary

 exists d, T {
  forall m {
   T[m] == sum n { (x[n] - mu[n]) * S[n,m] }
  } and
  d == sqrti( sum m { T[m] * (x[n] - mu[n]) } )
 } ;
```

Listing 7. Maximizing search CP from, for example, 458.sjeng.

```
domain ::
 N : int<32>
 move_ordering : int<32>[N]
range ::
 marker : int<32> = <0 to N-1>
relation ::
 i : int<32> = <0 to N-1>
 best : int<32>

 exists marker, best { move_ordering[marker] == best
  and best == max for i { move_ordering[i] } } ;
```

Listing 8. Minimizing search from 464.h264ref.

```
domain ::
 N : int<32>
 block_sad : int<32>[N]
 M : int<32>
range ::
 min_mcost : int<32>
 best_pos : int<32> = <0 to N-1>
relation ::
 pos : int<32> = <0 to N-1>

 exists best_pos, min_mcost {
  block_sad[best_pos] == min_mcost and
  min_mcost == min for pos { block_sad[pos] }
 };
```

Listing 9. Sum of absolute differences CP from 464.h264ref.

```
fun absi(x : int<32>) : int<32>

domain ::
 N : int<32>
 diff : int<32>[N]
range ::
 sad : int<32>
relation ::
 n : int<32> = <0 to N-1>
 exists sad { sad == sum n { absi(diff[n]) } } ;
```

Listing 10. Search CP as it can be found in various benchmarks.

```
domain ::
 N : int<32>
 element_bits : int<32>[N]
range ::
 found : bool<1>
relation ::
 n : int<32> = <0 to N-1>

 exists found {
  found == exists n { element_bits[n] == 0 }
 };
```

Listing 11. Sorting CP as found in 445.gobmk.

```
typedef ::
 t_moves : struct {
  score : int<32>
  pos : int<32>
 }
domain ::
 N : int<32>
 in : t_moves[N]
range ::
 out : t_moves[N]
relation ::
 n : int<32> = <0 to N-2>

 exists out { forall n { out[n].score <= out[n+1].score }
  and out >=< in } ;
```

Listing 12. Minimum-cost network flow CP from 429.mcf.

```
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>         //Primal feasible indicator, if the network is infeasible (and there is thus no satisfying
     flow/optcost) this is set to false
 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_conservation(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_conservation(flow) and flow_capacities
     (flow) { sum k { network.arcs[k].cost * flow[k] } } } } ;
```

Listing 13. CP for finding the Viterbi path as used in 456.hmmer or 482.sphinx3.

```
domain ::
 N : int<32>            //Number of states (excluding start=0/end=N+1 states)
 M : int<32>            //Number of observations (excluding start/end states
 a : int<32>[N+2,N+2]   //State transition probability matrix (including transitions from/to start/end states)
 x : int<32>[M]         //Observations
 p : int<32>[N+2,N+2]   //Emission probabilities, p[i,k] = P(Xi | Yk)
range ::
 y : int<32>[M+2]       //Likely sequence of states, including start/end states
relation ::
 P : int<32>[M+2]       //Likelyhood of likely states, including start/end states

 m : int<32> = <1 to M+1>
 l : int<32> = <1 to N+1>

 exists y { exists P { exists m {
  y[0] == 0 and         //First sequence is always start=0 state
  P[0] == 1 and         //Likelyhood to start in the start state = 1
  P[m] == p[x[m],y[m]] * a[y[m-1],y[m]] * P[m-1] and
  P[m] == max for l { p[x[m],l] * a[y[m-1],l] * P[m-1] }
 } } } ;
```