Fine-grained Benchmark Subsetting for System Selection

Pablo de Oliveira Castro[*†]
pablo.oliveira@uvsq.fr

Yuriy Kashnikov[†]
yuriy.kashnikov@exascale-computing.eu

Chadi Akel[†]
chadi.akel@exascale-computing.eu

Mihail Popov[†]
mihail.popov@exascale-computing.eu

William Jalby[*†]
william.jalby@uvsq.fr

[*]Université de Versailles Saint-Quentin-en-Yvelines, France
[†]Exascale Computing Research, France

ABSTRACT

System selection aims at finding the best architecture for a set of programs and workloads. It traditionally requires long running benchmarks. We propose a method to reduce the cost of system selection. We break down benchmarks into elementary fragments of source code, called codelets. Then, we identify two causes of redundancy: first, similar codelets; second, codelets called repeatedly. The key idea is to minimize redundancy inside the benchmark suite to speed it up. For each group of similar codelets, only one representative is kept. For codelets called repeatedly and for which the performance does not vary across calls, the number of invocations is reduced. Given an initial benchmark suite, our method produces a set of reduced benchmarks that can be used in place of the original one for system selection.

We evaluate our method on the NAS SER benchmarks, producing a reduced benchmark suite 30 times faster in average than the original suite, with a maximum of 44 times. The reduced suite predicts the execution time on three target architectures with a median error between 3.9% and 8%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques, Modeling Techniques; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement

1. INTRODUCTION

Benchmark suites are used to evaluate both compilers and architectures. Finding the best architecture and compiler optimizations for an application is an important problem for high performance computing, data centers, and embedded computers. Traditionally, before buying a new computing system, performance benchmarks are conducted to determine the best architecture and compiler options. This process involves running all the benchmark programs in different system configurations. This paper proposes a method to lower the cost of benchmarking by extracting a set of representative microbenchmarks sufficient to capture the performance characteristics of the original application.

Previous work [14, 23, 5] identifies many similarities among different programs in the same benchmark suite. We propose to take advantage of those similarities to reduce the benchmarking time.

We break down each application into small non-overlapping fragments of code, called *codelets*, using CAPS Entreprise's Codelet Finder (CF) tool [6, 1]. Our benchmark reduction method detects and removes redundant codelets. We address two sources of redundancy:

- **Similar computation kernels:** Benchmark suites share many similar codelets: simple ones, like set-to-zero or memory copy loops, and more complex ones, like Single-precision real Alpha X Plus Y (SAXPY) loops. There is no need to measure multiple copies of the same code.
- **Multiple invocations:** Codelets that are repeatedly invoked with the same context in an application lifetime, have the same running time for each invocation. In applications where a single codelet is called thousands of times, measuring only a few invocations achieves significant gains in benchmarking time.

Codelets are a key ingredient in our benchmark reduction strategy. Codelets allow us to break down a complex application into a set of small code fragments. By working at a fine granularity level we reveal much more redundancy. The larger a code fragment is, the harder it is to find a similar redundant fragment in another program. But, when codes are broken into elementary pieces, it is common to find duplicate computation patterns [2] such as dot products, array copies or reductions.

Our method first profiles the benchmark suite on a reference architecture. After this initial profiling, it produces a reduced set of microbenchmarks that can be reused on all the target architectures and configurations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CGO '14 February 15 - 19 2014, Orlando, FL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2670-4/14/02 ...$15.00.

The key idea is to cluster similar codelets together. Codelets from the same cluster share the same features and should react in the same way to architecture change. For example, memory-bound codelets will benefit from faster caches, whereas highly vectorized codelets will benefit from wider vectors. Therefore, by measuring a single representative per cluster we can extrapolate the performance of all its siblings.

Figure 1 presents the method composed of the following five steps:

Step A runs each application through CF hotspot detector to analyze the source code and divide it into a set of codelets.

Step B statically analyzes and profiles each codelet on the architecture chosen as a reference. We measure both static and dynamic features. Static features are extracted by the MAQAO static loop analyzer [9, 15] which provides detailed low-level performance metrics. Dynamic features are provided by Likwid 3.0 [28] which reads the hardware performance counters. Each codelet is tagged with a feature vector that gathers MAQAO and Likwid measurements. The feature vector is used as a performance signature to detect similar codelets.

Step C groups codelets sharing similar feature vectors into clusters.

Step D selects a representative for each cluster and extracts it as a standalone microbenchmark using CF extractor. The number of invocations in the microbenchmark is set as small as possible without degrading measurement accuracy.

Step E provides a model to predict performance of the original codelets from the measurements of the representatives.

Using this method we achieve significant speedups in benchmarking time. First, because only one benchmark per cluster is executed, and second, because it uses fewer invocations than the original application.

Next section discusses existing approaches to the problem and highlights our contributions. Section 3 details steps A to E of our methodology. Section 4 demonstrates our method on two different benchmark suites: a set of Numerical Recipes codes and the NAS SER benchmarks. Section 5 discusses the limitations of our approach and future work.

2. RELATED WORK

Our paper builds upon different works which fall into three broad categories:

- Techniques for code isolation or outlining
- Study of similarities among programs
- Benchmark reduction and subsetting

A first set of papers [1, 17, 22, 18] study code isolation. Lee and Hall [17] introduce the concept of code isolation for debugging and iterative performance tuning. Later works [22, 18] apply isolation techniques to automatic code tuning and specialization. Akel *et al.* [1] study under which conditions codelets preserve the performance characteristics of the original programs.

A second set of papers study similarities among programs, in order to uncover hidden redundancies or predict performance. Vandierendonck and Bosschere [29] analyze SPEC CPU 2000 execution time. They group applications according to their performance bottlenecks, showing that SPEC CPU 2000 contains redundant benchmarks. Hoste *et al.* [13, 12, 11] use microarchitecture-independent metrics to build a performance database that is used to predict performance of new programs. Phansalkar *et al.* [23] use a similar approach with hardware performance counters and statistical methods to analyze the redundancy of the SPEC CPU 2006 benchmark suite. Compared to the whole benchmark suite, 6 integer programs and 8 floating point programs capture the weighted average speedup with an error of 10% and 12% respectively. Bienia *et al.* [4] study redundancy between SPLASH-2 and PARSEC applications with statistical and machine learning methods They use execution-driven simulation with the Pin tool to characterize program's workloads and collect a large set of metrics, similar to the set used in [11] and [23]. They use Principal Components Analysis to improve the original feature space and hierarchical clustering to find redundancies between applications.

A third set of papers discusses benchmark reduction methods for speeding simulation time. Citron *et al.* [8] survey 173 papers from ISCA, Micro, and HPCA conferences and criticize methods that only use a subset of applications from the SPEC CPU benchmark suite. They demonstrate how projecting performance of a subset on the whole benchmark suite can bias speedups and yield incorrect conclusions. Contrary to existing approaches, our subsetting method does not remove entire applications from a benchmark suite, but only removes redundant fragments. Despite reducing the total time required for the benchmark suite evaluation, our method keeps the important performance information from the whole suite.

Lafage and Seznec [16] propose a method to find slices of a program that are representative for data cache simulation. It uses hierarchical clustering on two metrics: memory spatial locality and memory temporal locality. SimPoint [25, 26] is a tool which identifies similar program phases by comparing Basic Block Vectors (BBV). Phases are samples of 100M instructions. Simpoint reduces simulation time by removing repeated phases. BBV are program dependent, therefore SimPoint cannot use representatives of one program to predict another. In constrast, our method can take advantage of similarities across different applications. We show in Section 4.4 that exploiting inter-applications redundancies reduces the number of representatives while preserving accuracy. Eeckhout *et al.* [10] extend SimPoint by matching inter-application phases using microarchitecture-independent features.

SimPoint and its extensions are similar to our work in that they extract representative phases from an application. But SimPoint must be used in a simulator, whereas our method is more versatile since it produces C or Fortran codelets that can be recompiled and run both on simulators and on real hardware.

3. BENCHMARK REDUCTION STRATEGY

3.1 Codelet Detection

A codelet is a short fragment of source code without side-effects. Therefore, it can be outlined and extracted as a standalone microbenchmark. Multiple approaches have been presented for codelet identification and extraction [17, 22, 18]. In this paper we use the Codelet Finder tool (CF) [6] to outline C and Fortran outermost loops as codelets. Akel *et al.* [1] detail the CF identification and extraction process.

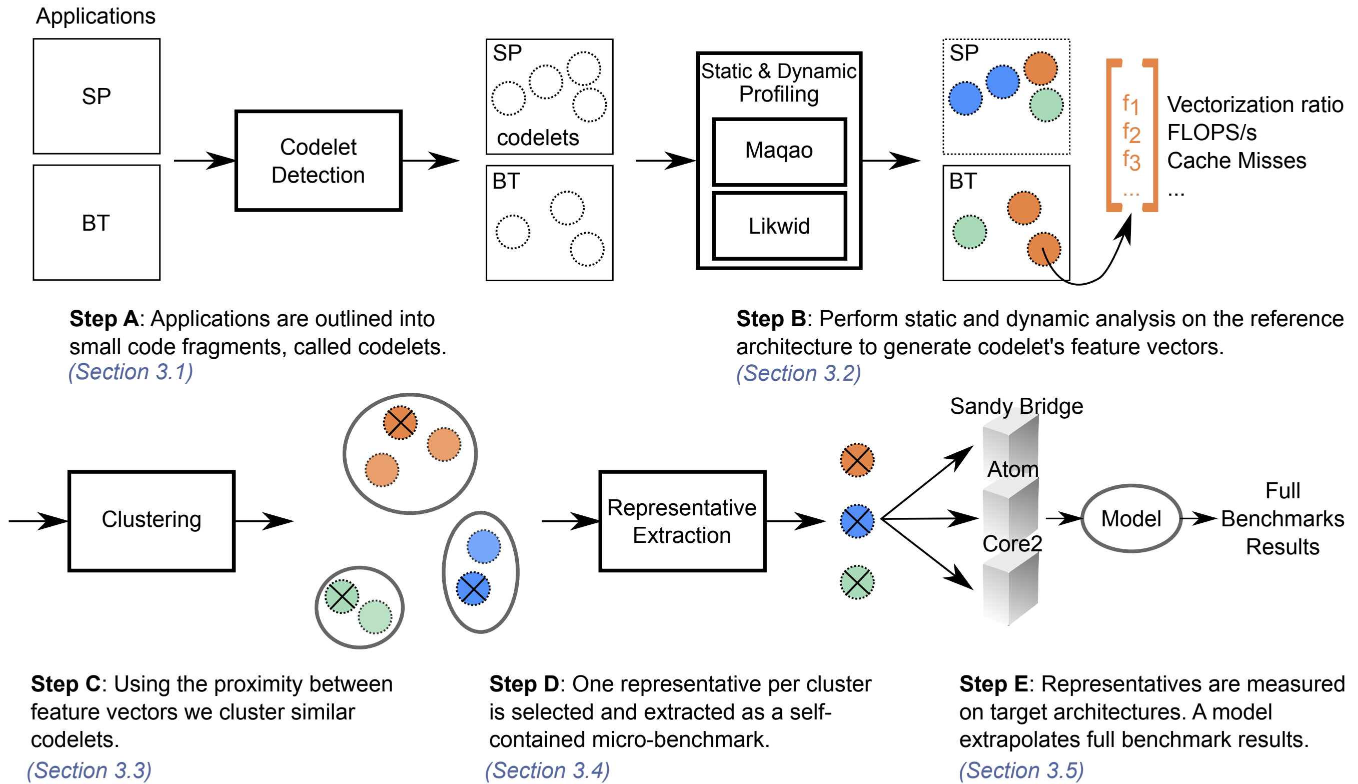


Figure 1: Overview of the benchmark reduction method

The first step of our benchmark reduction strategy is to use CF to identify the extractable codelets inside an application for future profiling. In the NAS SER benchmarks the detected codelets capture 92% of the execution time [1].

3.2 Profiling and Static Analysis

To detect similar codelets we measure performance features, both static and dynamic. Static features are useful to evaluate the assembly code quality and to detect performance problems specific to the microarchitecture. MAQAO static loop analysis [9, 15] provides a set of static metrics for the innermost binary loops. Examples of such metrics are the size of the loop, the pressure on dispatch ports, the number of used registers, the type of instructions. To compute these metrics, MAQAO disassembles and analyzes the binary. Some MAQAO metrics, provide a lower bound on performance by assuming that all the memory access hit the L1 cache. For benchmarks with datasets larger than L1, those metrics are less relevant. Hardware counters help us to overcome this issue and deal with dynamic hazards.

To characterize the dynamic behavior of codelets, we use a set of metrics provided by the Likwid tool [28]. Likwid measures hardware performance counters and derives a set of dynamic performance metrics such as execution time, cache misses, floating point instructions per second, or memory bandwidth.

To obtain accurate measurements, we automatically instrument the original source code enclosing each codelet between probes. The probes call Likwid's API. We discard codelets with execution time under one million cycles because they are too short to be accurately measured.

MAQAO and Likwid gather 76 different features. A subset of these features produce codelets' feature vectors. In section 4.2 we present how the feature subset is selected.

3.3 Clustering Similar Codelets

The feature vectors are the codelets performance signatures. Similar codelets share the same performance features and problems. The clustering step groups codelets with close feature vectors.

Let us consider N codelets named p_1 to p_N. Each codelet has an associated feature vector $\vec{f_i}$. Features are normalized to have unit variance and to be centered on zero. Normalization ensures that features have equal weight when computing a distance between two feature vectors. To evaluate the similarity of two codelets p_i and p_j we compute the euclidean distance between their feature vectors, $\|\vec{f_i} - \vec{f_j}\|$.

To cluster similar codelets, we use hierarchical clustering with Ward's criterion [30]. Hierarchical clustering is a greedy method. It starts with as many clusters as codelets. At each step, Ward's method merges a pair of clusters. The pair is selected to minimize the total within-cluster variance after the merge. The within-cluster variance of a cluster C_k is defined as the variance of all the feature vectors of codelets belonging to C_k. This variance is computed using the euclidean norm. Reducing the within-cluster variance forms compact clusters of codelets with close feature vectors. The clustering method ends when a single cluster is left. All the successive merges between clusters are recorded in a dendrogram. The final number of clusters, K, is selected by cutting the dendrogram at different heights. We offer two choices in our implementation:

- The user manually sets a number K of clusters.
- K is automatically selected using the Elbow method [27] which cuts the tree when the within-cluster variance stops improving significantly.

3.4 Extracting Cluster Representatives

We assume that by measuring a single representative per cluster we can estimate the performance of its siblings. A representative must adequately capture the performance features of its siblings: as a representative, we choose the codelet closest to the cluster centroid.

The selected representative is extracted as a standalone microbenchmark that can be compiled and run on the target architectures. CF [6] is able to produce a standalone executable for a given codelet. CF runs the original application and captures the memory accessed by the extracted codelet. The memory state is saved into a memory dump file. Then, CF generates a wrapper to build and execute the codelet as a standalone program. The codelet wrapper loads the memory dump file before a codelet is run to restore the original execution environment. Akel *et al* [1] detail the codelet extraction process.

To accurately time the standalone representative, we select a number of invocations so that the microbenchmark runs at least during 1 ms with a minimum of 10 invocations. We then take the median measurement among the invocations, to remove outliers.

The execution time of the microbenchmarks produced by CF generally matches the execution time of the original hotspot in the application. Yet for some *ill-behaved* codelets, the microbenchmark performance differs. Akel *et al.* [1] show that in the NAS benchmarks 19% of the codelets are ill-behaved. They fall into two categories:

- Codelets which are invoked with different contexts and various datasets during the lifetime of the application. CF captures only the dataset for the first invocation of a hotspot. Therefore, the microbenchmark only matches the first invocation inside the application.
- Codelets which are compiled differently inside and outside the application. Modern compilers use heuristics to measure the profitability and legality of an optimization before it is applied. When a codelet is extracted from the application, the code before and after the hotspot is not preserved. This may affect the optimizations that the compiler applies to the code.

The execution time of the representative is measured in the target architectures and is used to predict its siblings' performance. Therefore, it is very important for the representative to be *well-behaved*. We have implemented the following selection process:

1. Select representative.
2. Generate a stand-alone executable for the representative.
3. Compare the execution time for the representative and the original (profiled at step B). Representative is an ill-behaved codelet if its original and standalone execution differ by more than 10%.
4. If the representative is an ill-behaved codelet:
   - Mark ill-behaved representative as ineligible and start the selection process again.
   - If all the siblings are ineligible, the cluster is destroyed. Each ineligible codelet is moved to the cluster containing its closest neighbor.

If a cluster is only composed of ill-behaved codelets, the selection process reduces the number of clusters determined by the elbow method. Nevertheless, this selection guarantees that all representatives are well-behaved and therefore faithful.

3.5 Prediction Model

Because codelets from the same clusters have similar performance characteristics, we assume that they will similarly react to architecture changes.

Let the average execution time per invocation be t_i^{ref} for the reference architecture and t_i^{tar} for the target architecture. For codelet p_i, $s_i = t_i^{ref}/t_i^{tar}$ is the speedup between the reference and the target. We make the following assumption in our model: *codelets in the same clusters have the same speedup when moving to a new architecture.* In particular, the speedup of any codelet from cluster C_k, is close to s_{r_k}, the speedup of the cluster representative,

$$\forall p_i \in C_k, s_i \simeq s_{r_k}.$$

Therefore, measuring the set of representatives on the target architecture is enough to estimate the speedup of all the clusters. We predict the performance of each original codelet by using the following formula:

$$\forall p_i \in C_k, t_i^{tar} \simeq t_i^{ref} \times \frac{1}{s_{r_k}} = t_i^{ref} \times \frac{t_{r_k}^{tar}}{t_{r_k}^{ref}}.$$

These equations can be written in matrix form, $\vec{t_{all}^{tar}} \simeq \mathbf{M}.\vec{t_{repr}^{tar}}$, where $\vec{t_{repr}^{tar}}$ is the vector that contains the measurements of the representatives in the target architecture, $\vec{t_{all}^{tar}}$ are the predicted measurements for all the codelets, and $\mathbf{M}$ is a $N \times K$ matrix defined as,

$$\mathbf{M_{i,k}} = \begin{cases} 0 & \text{if } p_i \notin C_k \\ \frac{t_i^{ref}}{t_{r_k}^{ref}} & \text{if } p_i \in C_k \end{cases}$$

In this formulation, the matrix $\mathbf{M}$ represents the model that transforms representatives measurements into results for all benchmarks.

4. EXPERIMENTS AND VALIDATION

4.1 Experimental Setup

Table 1 presents the machines used for experiments. They belong to four different Intel CPU generations (Nehalem, Core 2 Duo, Atom, and Sandy Bridge) and possess quite distinct memory hierarchies. These machines were selected to validate that our benchmark reduction strategy is applicable on significantly different architectures. All the codelets are profiled on Nehalem, the reference architecture, at step B. The representatives are benchmarked on each target architectures (Atom, Sandy Bridge, and Core 2) at step E.

We evaluate our method using two criteria: the prediction error and the benchmarking reduction factor. For each

	Reference	Target		
	Nehalem	Atom	Core 2	Sandy Bridge
CPU	L5609	D510	E7500	E31240
Frequency (GHz)	1.86	1.66	2.93	3.30
Cores	4	2	2	4
L1 cache (KB)	4×64	2×56	2×64	4×64
L2 cache (KB)	4×256	2×512	3 MB	4×256
L3 cache (MB)	12	-	-	8
Ram (GB)	8	4	4	6

Table 1: Test architectures.

codelet, the prediction error is the difference between predicted and real measurements, computed as a percentage. The benchmarking reduction factor is the ratio between the execution times of the representatives and the full benchmark suite. These two metrics are tied. In practice, the more clusters we add, the more we decrease the prediction error. However, by adding more clusters we also increase the number of representatives and therefore the benchmarking time.

All benchmarks were compiled using Intel compiler `12.1.0` with the `-O3 -xsse4.2` (Nehalem and Sandy Bridge) and `-O3` (Core2 and Atom) optimization level. We use two benchmarks suites: 28 Numerical Recipes (NR) [24, 19] and 7 NAS SER benchmarks [3].

Each NR code is composed of a single computation kernel. There is a one-to-one mapping between the NR benchmarks and the NR codelets extracted. Moreover, all the NR codelets are well-behaved. NR codes are simple but cover a large spectrum of algorithms. They will be used as a training set to find a pertinent set of features. As described in section 3.3, the set of features is used to cluster together similar codelets. Not all the 76 features are relevant to performance benchmarking. To achieve accurate prediction, it is important to select pertinent features.

The NAS benchmarks are more complex than NR codes and produce 67 codelets. They are run with CLASS B datasets. The NAS benchmarks are used to validate that the feature set trained on NR can be successfully applied to more complex benchmarks. The next section describes how the feature set was selected on NR.

4.2 Feature Selection on Numerical Recipes

Irrelevant features add noise that degrades the clustering and the prediction accuracy. Therefore, it is important to wisely select features, keeping only those that adequately represent program behavior and improve prediction.

Evaluating the 2^{76} combinations of features is too costly. To find a good set of features in a reasonable time, we use genetic algorithms [31]. Genetic algorithms (GA) start with a population of randomly generated individuals. In our case, each individual represents a candidate feature set. This population evolves towards an optimal solution by recombining the best individuals with crossover and mutation operators.

An individual is encoded as a 76 boolean vector. The ith bit is set if and only if feature i is selected. For instance, the vector with all bits set to one, corresponds to the individual containing all the 76 features. Crossover and mutation operators are provided by the genalg [32] GNU R package.

To evaluate individuals, we consider the average prediction error of NR benchmarks on two architectures: Atom and Sandy Bridge. Best individuals should have a high prediction accuracy on both architectures but with a low number of representatives. To achieve this objective we choose the fitness function: $max(error_atom, error_sandybridge) \times K$ where K is the number of clusters. We intentionally leave Core 2 and NAS benchmarks out of the training process to fairly evaluate how our feature set fares on new architectures and new benchmarks.

We perform 100 GA iterations for a population size of 1000, and a mutation probability of 0.01. The Genetic algorithm converges to the optimal feature set presented in Table 2 by generation 47. The four selected dynamic features are computed using eight performance events that can be precisely measured in a single run without multiplexing.

Likwid dynamic features
- Floating point rate in MFLOPS.s^{-1}
- L2 bandwidth in MB.s^{-1}
- L3 miss rate
- Memory bandwidth in MB.s^{-1}
MAQAO static features
- Bytes stored per cycle assuming L1 hits
- Data dependencies stalls
- Estimated IPC assuming only L1 hits
- Number of floating point DIV
- Number of SD instructions
- Pressure in dispatch port P1
- Ratio between ADD+SUB/MUL
- Vectorization ratio for Multiplications (FP)
- Vectorization ratio for Other (FP+INT)
- Vectorization ratio for Other (INT)

Table 2: Best feature set found with a genetic algorithm evaluated with NR codelets on Atom and Sandy Bridge.

	K = 14		K = 24 elbow	
error	median	average	median	average
Atom	1.8%	12%	0%	1.70%
Sandy Bridge	3.2%	9.30%	0%	0.97%

Table 4: Prediction errors on Numerical Recipes with 14 and 24 clusters.

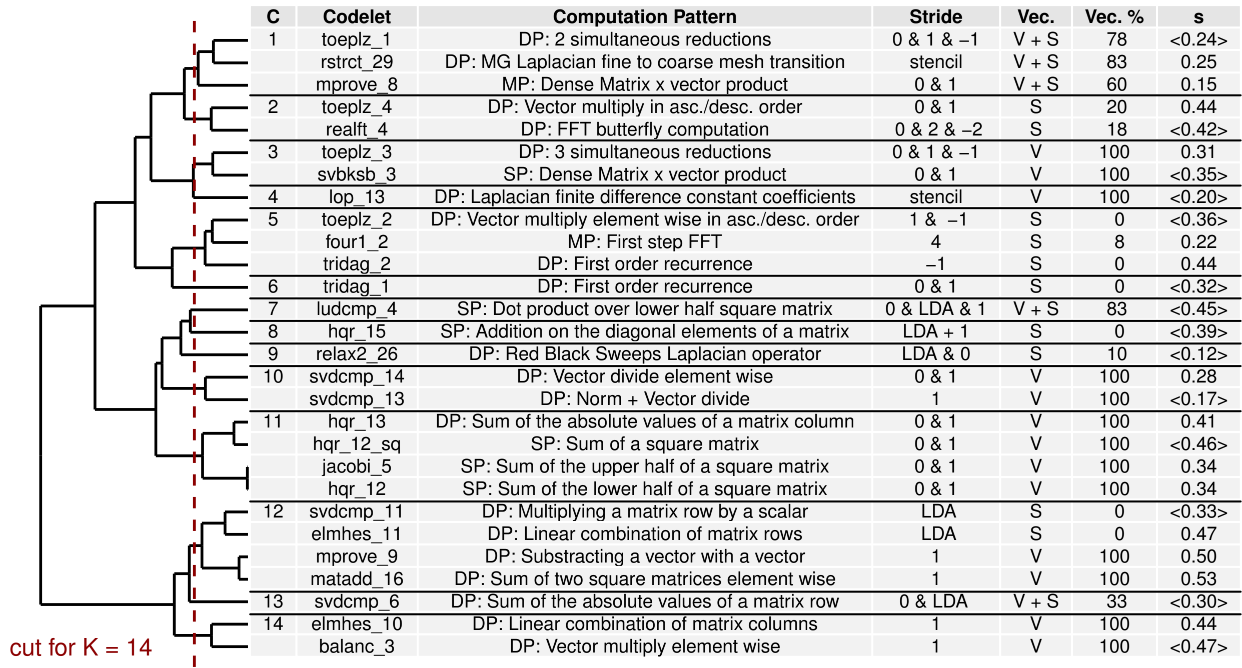
4.3 Numerical Recipes Evaluation

This section evaluates the clustering on NR codelets performed with the feature set described in table 2. Table 3 shows a 14-group clustering built on the reference architecture.

Despite depending on the reference architecture, our feature set is closely related to architecture-independent features [13, 12]. For example, codelets can use scalar instructions (S), vector instructions (V), or a mix of both (V + S). We manually analyzed the vectorization of the codelets, *Vec.*, and compared it to the vectorization ratio, *Vec. %*, reported by MAQAO. They are highly correlated.

Our assumption is that codelets in the same clusters should exhibit similar characteristics and behavior. An initial supporting observation is that the vectorization is homogeneous among clusters. We evaluate cluster similarity using two other similarity criteria.

First, we note that many clusters are formed of codelets with similar *computation patterns.* For example, cluster 10 gathers codelets that divide elements in a vector, cluster 11



C	Codelet	Computation Pattern	Stride	Vec.	Vec. %	s
1	toeplz_1	DP: 2 simultaneous reductions	0 & 1 & –1	V + S	78	<0.24>
	rstrct_29	DP: MG Laplacian fine to coarse mesh transition	stencil	V + S	83	0.25
	mprove_8	MP: Dense Matrix x vector product	0 & 1	V + S	60	0.15
2	toeplz_4	DP: Vector multiply in asc./desc. order	0 & 1	S	20	0.44
	realft_4	DP: FFT butterfly computation	0 & 2 & –2	S	18	<0.42>
3	toeplz_3	DP: 3 simultaneous reductions	0 & 1 & –1	V	100	0.31
	svbksb_3	SP: Dense Matrix x vector product	0 & 1	V	100	<0.35>
4	lop_13	DP: Laplacian finite difference constant coefficients	stencil	V	100	<0.20>
5	toeplz_2	DP: Vector multiply element wise in asc./desc. order	1 & –1	S	0	<0.36>
	four1_2	MP: First step FFT	4	S	8	0.22
	tridag_2	DP: First order recurrence	–1	S	0	0.44
6	tridag_1	DP: First order recurrence	0 & 1	S	0	<0.32>
7	ludcmp_4	SP: Dot product over lower half square matrix	0 & LDA & 1	V + S	83	<0.45>
8	hqr_15	SP: Addition on the diagonal elements of a matrix	LDA + 1	S	0	<0.39>
9	relax2_26	DP: Red Black Sweeps Laplacian operator	LDA & 0	S	10	<0.12>
10	svdcmp_14	DP: Vector divide element wise	0 & 1	V	100	0.28
	svdcmp_13	DP: Norm + Vector divide	1	V	100	<0.17>
11	hqr_13	DP: Sum of the absolute values of a matrix column	0 & 1	V	100	0.41
	hqr_12_sq	SP: Sum of a square matrix	0 & 1	V	100	<0.46>
	jacobi_5	SP: Sum of the upper half of a square matrix	0 & 1	V	100	0.34
	hqr_12	SP: Sum of the lower half of a square matrix	0 & 1	V	100	0.34
12	svdcmp_11	DP: Multiplying a matrix row by a scalar	LDA	S	0	<0.33>
	elmhes_11	DP: Linear combination of matrix rows	LDA	S	0	0.47
	mprove_9	DP: Substracting a vector with a vector	1	V	100	0.50
	matadd_16	DP: Sum of two square matrices element wise	1	V	100	0.53
13	svdcmp_6	DP: Sum of the absolute values of a matrix row	0 & LDA	V + S	33	<0.30>
14	elmhes_10	DP: Linear combination of matrix columns	1	V	100	0.44
	balanc_3	DP: Vector multiply element wise	1	V	100	<0.47>

Table 3: NR clustering with 14 clusters and speedups on Atom. The dendrogram on the left shows the hierarchical clustering of the codelets. The height of a dendrogram node is proportional to the distance between the codelets it joins. The dashed line shows the dendrogram cut that produces 14 clusters. The table on the right gives for each codelet: the cluster number C, the Computation Pattern, the Stride, the Vectorization, and the Speedup on Atom s. The speedup of the selected representative is emphasized with angle brackets.

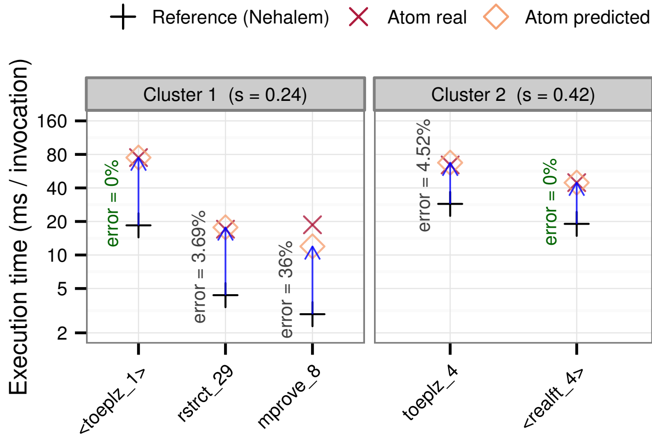


Figure 2: Predicted and Real execution times on Atom for clusters 1 and 2. Representatives are enclosed in angle-brackets. They have a 0% prediction error because they are directly measured. The representative speedup is applied to all its siblings to predict their target performance. Because the scale is logarithmic, applying the speedup is depicted by the arrow translation.

codelets that perform a reduced sum, cluster 14 codelets that compute element-wise multiplications on vectors or columns. The two "Dense Matrix x vector product" codelets have been separated because they use different floating point precision.

Second, the *stride* captures the distance between the data points accessed by two successive iterations of a codelet. For example, a stride of one means that the codelet is accessing memory sequentially. A stride of zero means an access to a constant memory location. A Leading Dimension Array (LDA) stride means a row-wise access to a column-wise stored array. If a codelet has two or more types of stride, we separate them with a '&' symbol. Stencil stride means that the kernel uses a five points stencil to access the data. Cluster 14 is composed only of codelets with contiguous access to memory. Cluster 11 contains only (0 & 1) codelets: one contiguous access to sweep the vector and one constant access for the accumulator. Other clusters have more complex stride behaviors.

The clustering is not perfect, but still gathers codelets sharing similar computation patterns, stride accesses, and vectorization.

Our second assumption, is that codelets with similar features have similar speedups on the target architectures. Column s on the table shows Atom speedups. The two codelets in cluster 10 suffer high slowdowns on Atom because they use high-latency division operations. Our feature set captures this pattern and isolates them in their own cluster.

In most of the clusters, speedups are homogeneous. Close codelets in the dendrogram such as in clusters 2, 3, or 14 exhibit close speedups. Yet in some clusters such as 10 or 12

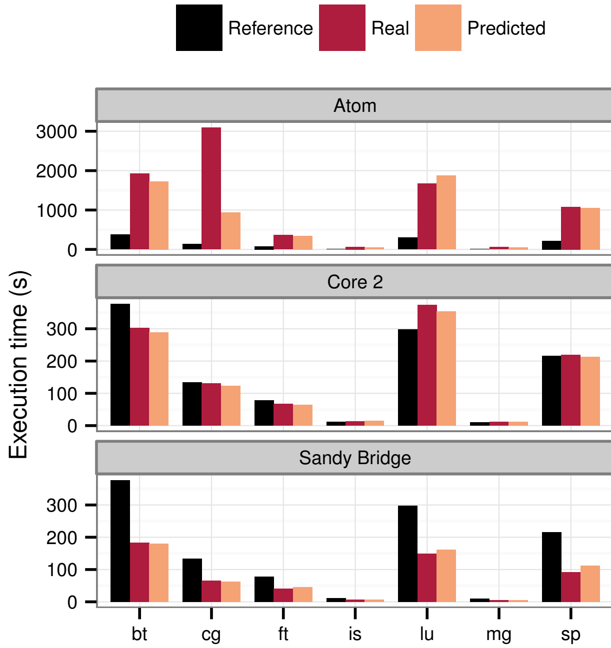


Figure 5: Predicted and Real execution times on the target architecture compared to the execution time on the reference architecture.

Reduction	Total	Reduced invocations	Clustering
Atom	44.3	×12	×3.7
Core 2	24.7	×8.7	×2.8
Sandy Bridge	22.5	×6.3	×3.6

Table 5: Benchmarking reduction factor breakdown with 18 representatives.

the speedups are distinct. Our dendrogram cut is too rough and a higher number of clusters is needed. In this case, 24 clusters, as recommended by the elbow method, fix the most striking discrepancies. Yet the 24 elbow clustering, though more conservative in terms of prediction, is less interesting to analyze because it has many singleton clusters.

We evaluate the prediction error using the 14 clusters' representatives on Atom and Sandy Bridge. Figure 2 details the prediction model for the cluster 1 and 2. As expected, the representatives codelets `toeplz_1` and `realft_4` are perfectly predicted. The prediction is accurate except for `mprove_8`. Table 3 dendrogram shows that slightly increasing K puts the offending codelet in a different cluster.

Table 4 summarizes the prediction errors for all the NR codelets on Atom and Sandy Bridge. The overall accuracy of the prediction is good. Nevertheless, the NR were used during the feature selection training. The feature set was selected to minimize prediction accuracy. Next section validates our method on a different set of benchmarks and one new architecture not used during training.

4.4 Subsetting the NAS Benchmark Suite

In this section we reuse the feature set trained on NR benchmarks and validate our benchmark reduction method on the NAS SER suite. We also evaluate a new target architecture Core 2. We show that the number of clusters

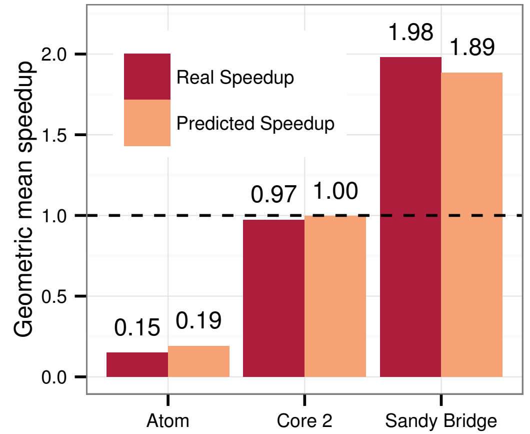


Figure 6: Geometric mean speedup per architecture.

selected by the elbow method provides a good trade-off between benchmarking reduction factor and prediction error. Then we analyze codelets and applications prediction.

Benchmarking reduction versus prediction error.

Figure 3 shows the trade-off between prediction error and benchmarking reduction factor while we increase the number of clusters. As expected the more clusters we add, the lower the median error becomes. On the other hand, the benchmarking reduction becomes less effective because we have more codelets to run on the target architecture. The dashed line in figure 3 marks the number of clusters chosen by the elbow method, here 18. If needed, the user can tune the number of clusters depending on what he wants to optimize: faster benchmarking or lower prediction error.

The elbow clustering achieves a high reduction factor between 23 and 44, while maintaining a low prediction error between 3.9% and 8%. Unsurprisingly, Atom, the most different architecture from the reference, has the highest prediction error.

The benchmarking reduction comes from two factors. First, representatives are benchmarked during a small number of invocations as described in section 3.4. Second, by clustering the codelets, only the representatives have to be measured. Table 5 breaks down the contributions of the two factors. The reduction due to clustering is close to the ratio between the original number of codelets and the number of representatives, $\frac{67}{18} \approx 3.7$. It is not exactly the same, because some codelets are longer than others.

Codelet performance prediction.

Figure 4 shows the predicted and real execution times on Sandy Bridge. The boxes gather the codelets by application. The applications may contain codelets coming from different clusters with different speedups. The execution time on Sandy Bridge is predicted with a median error of 5.8%. The error mainly comes from short-lived codelets (less than 10 ms per invocation) which are more affected by measurement errors such as instrumentation overhead. Codelets are faster on Sandy Bridge than on the reference. It is not surprising as Sandy Bridge frequency is almost twice the reference one. The median prediction error is 8% for Atom and 3.9% for Core 2.

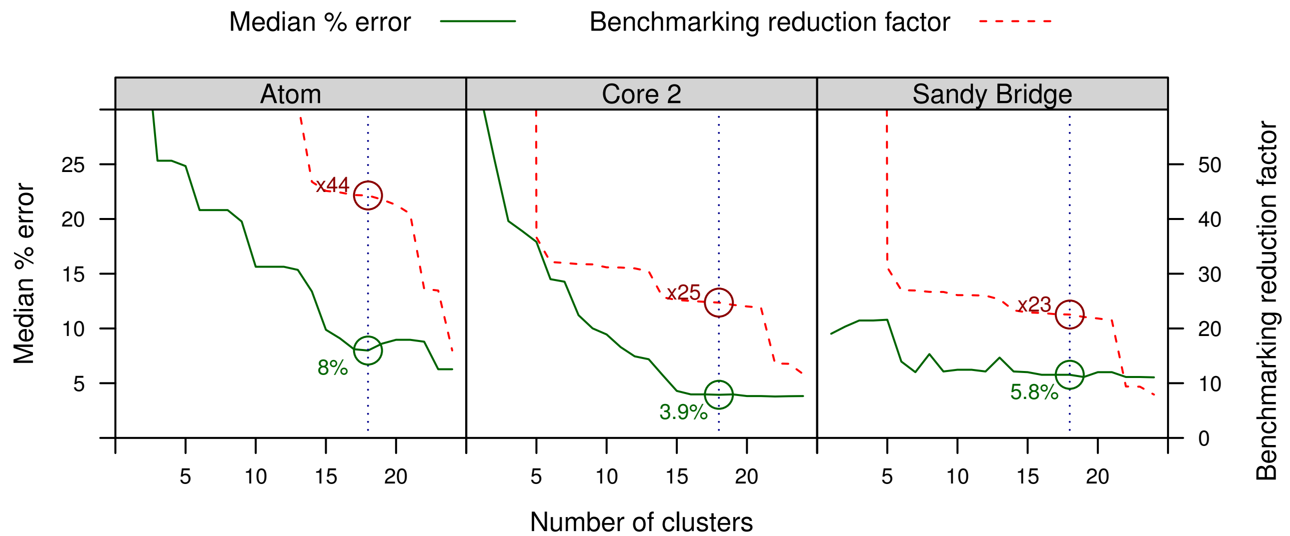


Figure 3: Evolution of prediction error and benchmarking reduction factor on NAS codelets as the number of clusters increases. The dotted vertical line marks 18, the number of clusters selected by the elbow method.

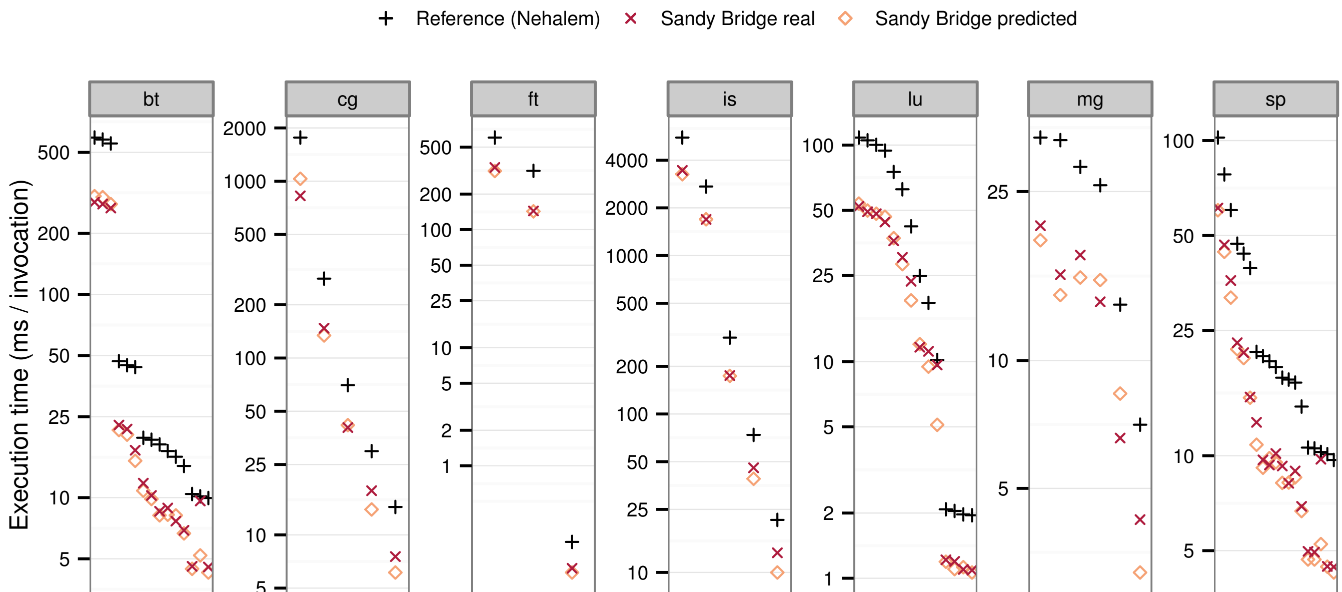


Figure 4: Predicted and Real execution times on Sandy Bridge compared to the Nehalem reference execution. Each box presents the codelets extracted from one of the NAS applications. Only three codelets in BT, LU, and SP are mispredicted.

Application performance prediction.

Codelets capture 92% of the execution time of the original NAS applications [1]. Therefore, by aggregating the individual codelet predictions, we accurately predict the original applications performance.

The whole application prediction is done in two steps. First, we estimate the speedup of the part of the application covered by codelets. The application's codelets predictions are aggregated and weighted by their number of invocations. Second, we assume that the speedup of the unknown part of the application is equal to the one of the covered part.

Figure 5 shows the application prediction on the three target architectures. Atom is significantly different from the reference: it is an in-order processor, without L3 cache, nor SSE4 vector instructions. Moving to Atom slows down all the benchmarks. The prediction accuracy on Atom is high, except for the Conjugate Gradient (CG) benchmark. CG's huge error is caused by a single codelet representing 95% of its execution time. This codelet is well-behaved on Nehalem and is selected as the representative. Yet, on Atom the extracted microbenchmark is much faster than the original codelet and incurs 1.6 times less cache misses. The microbenchmark is not preserving the cache state. This behavior was only observed on Atom.

On Sandy Bridge, all the applications are faster. Sandy Bridge has the fastest frequency and a more modern microarchitecture than the reference. The prediction accurately captures the speedups for all the original applications.

Core 2 microarchitecture is older than our reference, yet it has a higher frequency. The performance between both architectures is very close, providing an interesting challenge for system selection. Indeed, the best architecture depends on the application of interest. Some applications are faster, like BT and FT, while other are slower, like LU. Our reduced benchmark set captures this behavior and correctly predicts the trend allowing the user to smartly select the best architecture depending on the application.

In order to evaluate the overall benefits of an architecture, we compute the geometrical mean of the applications speedups. Figure 6 shows the predicted and the real speedup for each architecture. The reduced benchmarks accurately predict the expected speedup for each architecture.

Capturing architecture change.

To illustrate how our method captures architecture change, we consider two of the 18 clusters. Cluster A contains two codelets, `LU/erhs.f:49-57` and `FT/appft.f:45-47`. Both are a triple-nested loop with high latency operations such as division and exponential. They are computation bound. Cluster B also contains two codelets, `BT/rhs.f:266-311` and `SP/rhs.f:275-320`. Both are computing a three-point stencil on five planes. Cluster B codelets are memory bound.

Our features correctly separate the two performance patterns: static IPC is high in cluster A whereas memory and cache bandwidths are high in cluster B. The compute bound cluster A is 1.37 times *faster* on Core 2 due to higher clock frequency. On the contrary, the memory bound cluster B is 1.34 times *slower* on Core 2 because the last-level cache is four times smaller than the reference. The clustering correctly separates the two behaviors producing an accurate prediction for both groups.

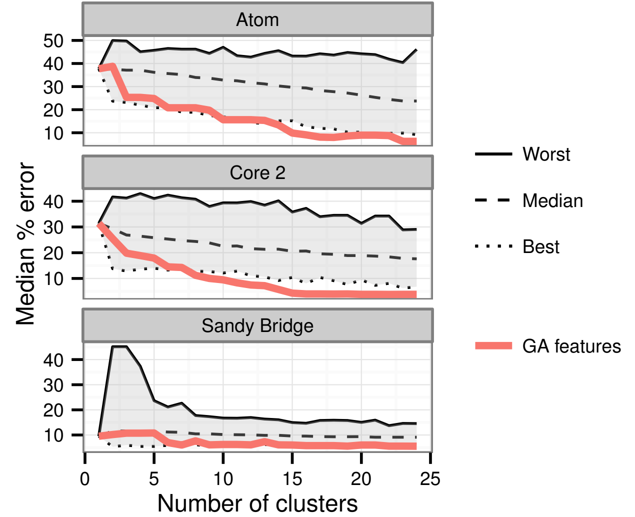


Figure 7: Genetic-Algorithm feature clustering compared to random clustering. For each number of clusters, from 2 to 24, 1000 random clusters are evaluated. Clustering with our GA feature set is consistently close or better than the best random clustering (out of 1000).

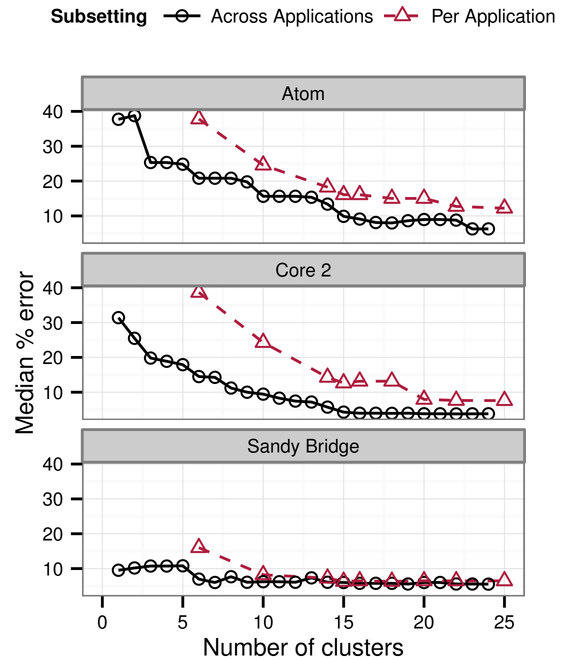


Figure 8: Sharing representatives *across* applications is more efficient than *per-application* representatives. Shared representatives can exploit inter-application redundancy, achieving low prediction errors with less representatives.

Evaluation of the feature-guided clustering.

To evaluate the quality of our feature-guided clustering, we compare it to 1000 random clusterings. In Figure 7, we make K, the number of clusters, vary from 1 to 24. For each value of K, we generate 1000 random partitionings into K clusters. We compute the prediction error for each partitioning after applying steps D and E. The proposed feature-guided clustering is most of the time close or better than the best random clustering. Our choice of features and clustering yields competitive results.

It is difficult to compare SimPoint [25, 26] and our method directly, because SimPoint only runs in a simulator. Nevertheless, SimPoint uses Basic Block Vectors (BBV) to cluster phases. Because BBV are application dependent, SimPoint cannot use representatives of one program to predict another. On the contrary, our approach is able to share representatives across a set of applications. We evaluate the benefits of subsetting across applications by comparing our results to a subsetting where different applications do not share representatives.

To simulate per-application subsetting, we execute Steps A to E on each application *separately*, and aggregate the results. The number of representatives is distributed evenly among the separate applications. The number of representatives per application, varies from one up to twelve. MG cannot be predicted with the per-application subsetting because its codelets are ill-behaved. Therefore, MG was excluded from the per-application error computation.

Figure 8 shows that subsetting across applications achieves high accuracy with less representatives because it exploits inter-application redundancy. Eeckhout *et al.* [10] reach similar conclusions.

5. LIMITATIONS AND FUTURE WORK

Our methodology successfully accelerates benchmarking on a wide range of Intel architectures. In this section we discuss some of its drawbacks.

Overhead of reducing the benchmark suite.

Profiling the benchmarks on the reference architecture and extracting the representatives is a costly process. For instance, extracting the 18 codelets into microbenchmarks from the NAS suite takes 380 minutes. If the user is only interested in a single architecture, our method does not pay off: it is quicker and more accurate to fully run the benchmarks once. Yet, when comparing many target architectures, typically for system selection, the overhead of profiling and extracting the representatives is quickly amortized. Also, the benchmarks are portable, so they can be extracted once for a benchmark suite and reused by many different users.

Feature Set.

Some of the features we consider are architecture-dependent. When the reference and target architectures are of the same family, as in this paper, the considered features are pertinent. They capture the different architecture performance bottlenecks, and produce accurate predictions. However, applying our method to a completely different architecture such a GPU, may require some extensions. Architecture independent metrics [13, 12] could generalize our method.

We would also like to extend our method to parallel applications by extending the set of considered metrics to capture synchronization and communication bottlenecks [7].

6. CONCLUSION

This paper presents a methodology that significantly reduces benchmarking time and applies it to the system selection problem. Unlike previous work that selects a subset of benchmarks at the application level, our method operates at a finer level by extracting fragments of code, called codelets. By detecting similar codelets within an application or across different applications, we reduce a full benchmark suite to a small set of representatives microbenchmarks.

On the NAS benchmarks, our methodology reduces the benchmarking time up to 44 times with a prediction error under 8%. The reduced subset accurately predicts the original application speedups. It finds the best architecture for each application at a low benchmarking cost.

The extracted microbenchmarks are portable source-code snippets. Our method could be extended to other contexts such as compiler regression test-suites or auto-tuning.

The data and code used in this paper are available as an IPython Notebook [21, 20] that allows to reproduce our experiments. The notebook can be accessed at `http://benchmark-subsetting.github.io/fgbs/`.

Acknowledgements

The authors would like to thank Florent Conti for his insightful comments and his help producing Figure 7.

This work has been conducted by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, and UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

7. REFERENCES

[1] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby. Is Source-code Isolation Viable for Performance Characterization? In *Parallel Processing Workshops (ICPPW), 2013 42nd International Conference on*. IEEE, 2013.

[2] M. Arenaz, J. Touriño, and R. Doallo. Xark: An extensible framework for automatic recognition of computational kernels. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):32, 2008.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991.

[4] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56. IEEE, 2008.

[5] R. Cammarota, A. Kejariwal, P. D'Alberto, S. Panigrahi, A. V. Veidenbaum, and A. Nicolau.

Pruning hardware evaluation space via correlation-driven application similarity analysis. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, page 4. ACM, 2011.

[6] CAPS entreprises. Codelet finder.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 2–12. IEEE, 2013.

[8] D. Citron. MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 52–61. ACM, 2003.

[9] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, W. Jalby, et al. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[10] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 2–12. IEEE, 2005.

[11] K. Hoste and L. Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 83–92. IEEE, 2006.

[12] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *Micro, IEEE*, 27(3):63–72, 2007.

[13] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 114–122. ACM, 2006.

[14] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *Computers, IEEE Transactions on*, 55(6):769–782, 2006.

[15] Y. Kashnikov, P. de Oliveira Castro, E. Oseret, and W. Jalby. Evaluating architecture and compiler design through static loop analysis. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 535–544. IEEE, 2013.

[16] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload characterization of emerging computer applications*, pages 145–163. Springer, 2001.

[17] Y.-J. Lee and M. Hall. A code isolator: Isolating code fragments from large programs. In *Languages and Compilers for High Performance Computing*, pages 164–178. Springer, 2005.

[18] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*, pages 308–322. Springer, 2010.

[19] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler. Simsys: a performance simulation framework. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, page 1. ACM, 2013.

[20] F. Pérez and B. E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[21] F. Perez, B. E. Granger, and C. P. S. L. Obispo. An open source framework for interactive, collaborative and reproducible scientific computing and education. 2012.

[22] E. Petit, G. Papaure, F. Bodin, et al. Astex: a hot path based thread extractor for distributed memory system on a chip. In *Proceedings of Compilers for Parallel Computers workshop (CPC2006)*, 2006.

[23] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 412–423. ACM, 2007.

[24] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes: The art of scientific computing*. Cambridge university press, 1986.

[25] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 3–14. IEEE, 2001.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 45–57. ACM, 2002.

[27] R. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[28] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[29] H. Vandierendonck and K. De Bosschere. Many benchmarks stress the same bottlenecks. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2004.

[30] J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[31] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[32] E. Willighagen. GNU R package ‘genalg’, 2013.