MB3 D6.2– Report on regions of interest as mini application candidates
Version 1.0

Document Information

Contract Number	671697
Project Website	www.montblanc-project.eu
Contractual Deadline	PM12
Dissemination Level	Public
Nature	Report
Authors	Lavanya Ramapantulu (UVSQ)
Contributors	Pablo de Oliveira Castro (UVSQ)
Reviewers	Gabor Dozsa (ARM) Jesus Labarta (BSC)
Keywords	Performance analysis, Benchmark subsetting, dynamic features, static features, clustering, accelerators, vectorization ratio

Notices: *This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N^o 671697.*

Change Log

Version	Description of Change
v0.1	Initial version of the deliverable
v0.2	Incorporated suggestions and fixes by Jesus Labarta (BSC)
v0.3	Incorporated suggestions and fixes by Gabor Dozsa (ARM)
v1.0	Final version

Contents

Executive Summary

This document presents a methodology to select interesting regions within applications that exhibit variety with respect to processor resource demands and are representative of a set of benchmark applications. Such a selection of a benchmark subset allows for piecewise optimization of an application by replaying the selected regions on a simulator in contrast to executing all the applications thus reducing simulation time.

Firstly we describe briefly the chosen applications from PARSEC and lulesh, and then present the detailed approach to select the regions. Next, we apply the proposed approach and show how the selected regions can be used for benchmarking novel accelerators, and performance tuning of big and LITTLE processors in heterogeneous architectures.

1 Introduction

The following document reports on the selection of regions of interest within applications to achieve two objectives. Firstly, with the advent of heterogeneity in processors like the big.LITTLE architecture from ARM [Jef13], it is increasingly important to match the application's demands to the computational resources. Secondly, cycle accurate simulations are expensive in terms of the simulation time ranging from couple of hours to days [ESC05, PHC03]. Furthermore many applications exhibit similar computational resource demands [CAP$^+$15]. Thus, there is a need for a methodology to identify interesting regions within applications that exhibit diverse resource demands so that these representative regions can be used for (i) simulating new architectures and considerably reducing simulation time, and (ii) determining a good match between application's demands and the available heterogeneous resources such as processors or accelerators. This work is part of WP6 looking at application's co-design and attempts to capture patterns within applications that can be used for the runtime/architecture evaluation without executing the full application. This will also aid in the design of new accelerators by providing interesting application regions for their performance evaluation.

2 Methodology for Region Selection

The objective is to develop an approach to select regions of interest within applications that are representative of both variety in terms of system resource demands and are similar to other regions across and within applications to reduce simulation time. An overview of the proposed approach is shown in Figure 1. Given an application mix, the proposed approach first derives dynamic features from them using innermost loops as a basic block for a region. In addition to the dynamic features that determine run-time demands of the program from system resources such as memory, the proposed approach also uses static features that are inherent to the program itself. Using both these feature sets, we use a clustering method to classify and group similar regions and then derive representative regions from each cluster.

Innermost loops are pinpointed at the binary level after compiler optimizations such as loop unroll or loop interchange have been applied. The analysis is performed using UVSQ's MAQAO tool suite which directly operate at the binary level. The original source code of the applications is not required to perform the region selection.

The next sections detail each of the steps in the methodology from choosing the application mix, selecting the features and the clustering technique used.

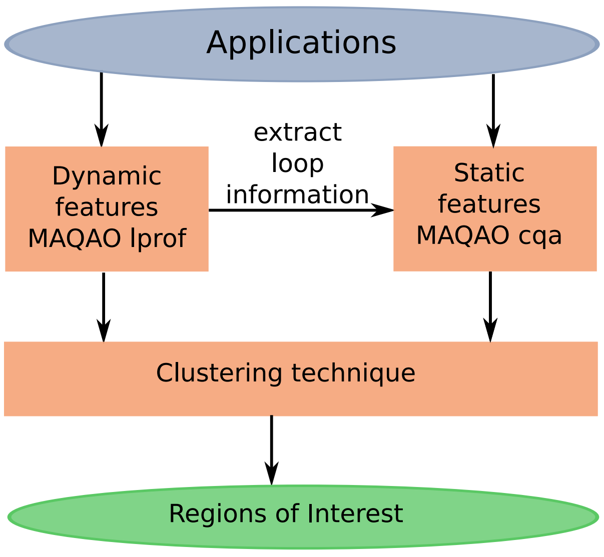

Figure 1: Approach overview

2.1 Applications

Our methodology is applied to select regions of interests within applications that meet the dual-goal of variety and representativeness. To meet the goal of variety, we need to choose benchmarks that exert diverse performance demands with respect to system resources such as cores and memory. Thus, we choose a diverse set of applications to as an input to our methodology among which we apply the approach to select regions of interest. These diverse set of applications are from two open-source code sets. We use 13 applications from the PARSEC benchmark suite [Bie11] and the lulesh code [KKN13] to apply our methodology to a proxy application.

2.1.1 PARSEC

In this section, we describe the 13 programs that we use from the PARSEC suite. The description in this section is taken from the sources [BKSL08], [BL09] and [Bie11] by Bienia et al.

blackscholes This application uses the Black-Scholes partial differential equation [BS73]

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is the price of an option and a function of the stock price S with volatility σ at time t if the constant interest rate is r. This equation is used to calculate the prices for a portfolio of European options and the prices are computed numerically as there is no closed-form solution for the Black-Scholes equation. This application is bounded by the number of floating-point computations performed by a computer.

bodytrack This is a computer vision application using multiple cameras to track a human body with image sequences. It uses computer vision algorithms to extract these images from the streaming video. It uses an annealed particle filter to track the movement of the body and pin down the exact location of the body and posture on the image from the video stream. More details about it's implementation is found at [BKSL08].

canneal Canneal is representative of a kernel that has high demands with respect to memory accesses. This kernel uses the simulated annealing algorithm to determine the routing within chips and minimize this cost. It is widely used in computer aided design tools for large processor designs. This minimization method swaps the routes within the processor circuit elements in a psuedo-random manner and accepts the swap if the routing is minimized. Sometimes a routing that increases the cost is also accepted to escape from local minima. The algorithm converges when the number of swaps decreases to a stable value. A more detailed description of the algorithm is presented in [Ban94].

dedup This kernel uses a combination of global and local compression called "deduplication" to compress a data stream and achieve high compression ratios. This method of compression has applications in mainstream backup storage systems [QD02] and hence is a useful benchmark to include in this study. This kernel uses five pipeline stages and the first stage reads the input stream and breaks it into coarse-grained chunks. The second stage uses rolling fingerprinting to obtain fine-grained segments from these coarse-grained chunks. For each of these segments, the third stage computes a hash-value which in turn is used by the fourth stage to compress these values using the Ziv-Lempel algorithm. The fifth and final stage then assembles these compressed values and hash values into the deduplicated output.

facesim This application employs physical simulation to compute a realistic animation of the modeled face. With the increase in usage of computer games, this is a useful application as it creates a more realistic virtual environment. It uses three kernels for computing the state of the face mesh at the end of each iteration. The first kernel determines the steady state of the simulated mesh by using the Newton-Raphson method to solve the nonlinear system of equations. The second kernel iterates over all the tetrahedra of the mesh and determines the velocity-independent forces acting on the simulation mesh. The third kernel solves the system of linear equations from the two kernels using the conjugate-gradient algorithm.

ferret This application does image similarity search and is based on the Ferret toolkit [LJW+06]. This is relevant as it represents the emerging Internet search engines for non-text data. It uses six stages, with the first and last stage processing input and output respectively. The middle four stages are, (i) segmentation of the query image, (ii) extracting features, (iii) indexing the image database with candidate sets and (iv) ranking the database by computing a detailed similarity estimate and ordering the database accordingly.

fluidanimate This application simulates an incompressible fluid for interactive animation purposes by extending the Smoothed Particle Hydrodynamics (SPH) method. At each time-step this application uses five kernels for computing the simulation, namely (i) rebuild spatial index, (ii) compute densities, (iii) compute forces, (iv) handle collisions with scene geometry and (v) update positions of particles. With the increasing adoption of physical simulation in real-time animations and computer games, this application provides a good example representation.

raytrace This benchmark renders a 3D scene so that it can be seen on the screen by a human observer. The basic idea of the ray tracing method is to shoot rays into a scene and compute where they hit objects. A new set of rays is then created at each intersection point to simulate effects such as reflections and refractions. To accelerate this process ray tracers usually use a data structure that is called a Bounding Volume Hierarchy (BVH). A BVH organizes the entire scene in a tree structure, which means that by descending down from the root ray tracers can find ray-surface intersection points extremely fast. A more detailed description of the raytrace workload with its core algorithms and data structures can be found in [BL09].

streamcluster The streamcluster application computes a predetermined number of medians for a given stream of input points, such that each point is assigned to its nearest center. The sum of squared distances metric is used to determine the nearest center and this benchmark represents the organization of large amount of continuously streaming data in real-time such as in data mining, or network intrusion detection. This application is memory intensive when the dimensionality of the incoming data is low and it becomes computationally bound as the dimensions increase.

swaptions This application determines the price of a portfolio of swaptions using the Heath-Jarrow-Morton (HJM) framework [HJM90]. For a given class of models, this framework is useful to determine the evolving interest rates for asset liability and risk management. As these models are non-Markovian, price cannot be determined by solving the partial-differential equations and thus is different from the blackscholes application. Therefore, this application uses Monte-Carlo simulations to determine the price.

vips This application is based on the VASARI Image Processing System (VIPS) with the benchmark derived from the print on demand service at the national gallery of London. It includes typical image operations such as affine transformations and convolutions. The image transformation step has 18 stages and is grouped into the following modules. The first module is a crop step that removes 100 pixels from all edges. The next module is the shrink operation that reduces the image by 10% and uses bilinear interpolation to compute the output. The next module adjusts the white points and shadows to improve visual quality of the perceived image. The last module sharpens the image by exaggerating the edges using a Gaussian blur filter.

x264 This application is an Advanced Video Coding (AVC) video encoder and is based on the ITU-T H.264 standard which is also the ISO/IEC MPEG-4. This improves over previous standards by increasing the precision of the sample bit depth, using colours with higher resolution and context adaptive binary arithmetic coding (CABAC). These improvements enhance the quality of the output of H.264 encoders and hence are used in a wide-range of systems from video conferencing equipment to high-definition movies.

Program inputs Table 1 presents the summary of the PARSEC benchmarks used in this report along with their input size.

Program	Domain	Problem Size	
		simlarge	**native**
blackscholes	Financial Analysis	65,536 options	10,000,000 options
bodytrack	Computer Vision	4 frames, 4000 particles	261 frames, 4000 particles
canneal	Engineering	400,000 elements	2,500,000 elements
dedup	Enterprise Storage	184 MB data	672 MB data
facesim	Animation	1 frame, 372,126 tetrahedra	100 frames, 372,126 tetrahedra
ferret	Similarity Search	256 queries, 34,973 images	3,500 queries, 59,695 images
fluidanimate	Animation	5 frames, 300,000 particles	500 frames, 500,000 particles
freqmine	Data Mining	990,000 transactions	250,000 transactions
raytrace	Rendering	3 frames, 1920 × 1080 pixels	200 frames, 1920 × 1080 pixels
streamcluster	Data Mining	1 block, 16,384 points per block	5 blocks, 200,000 points per block
swaptions	Financial Analysis	64 swaptions, 20,000 simulations	128 swaptions, 1,000,000 simulations
vips	Media Processing	1 image, 2662 × 5500 pixels	18,000 × 18,000 pixels
x264	Media Processing	128 frames, 640 × 360 pixels	512 frames, 1920 × 1080 pixels

Table 1: Summary of PARSEC applications

2.1.2 lulesh

This section is taken from the sources, online [GPU], online [Lab] and the [LUL] report.

Computer simulations of a wide variety of science and engineering problems require modeling hydrodynamics, which describes the motion of materials relative to each other when subject to forces [HKG11]. Many important simulation problems of interest to DOE involve complex multi-material systems that undergo large deformations. LULESH is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications.

LULESH represents a typical hydrocode and approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. LULESH is built on the concept of an unstructured hexahedral mesh with two centerings [K+12]. The element centering (at the center of each hexahedral) stores thermodynamic variables, such as energy and pressure. The nodal centering (where the corners of hexahedrals intersect) stores kinematics values, such as positions and velocities. The simulation is run via a time stepping algorithm followed by a time constraint calculation. The algorithm consists of two major steps: advancing the node quantities, followed by advancing the element quantities. Advancement of the node quantities requires calculating the nodal forces, which is the most compute intense part of the simulation. To advance element quantities first kinematic values are calculated for the elements based on the new nodal positions and velocities. We have used an input size of 50^3 for measuring the features.

2.2 Feature Selection

To meet the goal of selecting regions of interest, there is a plethora of metrics to choose from and this choice directly impacts the variety in the selected regions. To cater to the increasing adoption of heterogeneity in the underlying architecture systems and the use of accelerators, we select a range of dynamic and static features to capture both system level and program level characteristics respectively.

2.2.1 Dynamic Features

The goal is to select a set of metrics that represent variety in terms of system resource demands during execution. As the methodology needs to adapt to different systems being simulated, we have the conflicting goal of selecting features that represent variety in terms of system usage but at the same time the selected features should not be too dependent on the underlying system characteristics. Thus, we abstract the performance metrics of the system in a coarse-grained manner to achieve both variety and some level of independence from the underlying micro-architecture.

To extract these performance features, we use the MAQAO toolchain of UVSQ [DBT+07]. This toolchain provides both static and dynamic analysis tools and users are able to characterize the behaviour of programs at both function and loop level. We use loop level characterization to identify the regions of interest, and use MAQAO *lprof* to measure the dynamic features. MAQAO *lprof* uses value profiling at the assembly level and thus causes minimum overhead to the actual run-time of the application. The hardware counters that are profiled are UNHALTED_CORE_CYCLES, INST_RETIRED, BRANCH_INST_RETIRED, BRANCH_MISS_RETIRED, LLC_REFERENCES and LONGEST_LAT_CACHE:MISS. These six values are in turn used to compute the dynamic features, CPI, BMR and LMR using:

$$CPI = \frac{UNHALTED\_CORE\_CYCLES}{INST\_RETIRED}$$
$$BMR = \frac{BRANCH\_MISS\_RETIRED}{BRANCH\_INST\_RETIRED}$$
$$LMR = \frac{LONGEST\_LAT\_CACHE : MISS}{LLC\_REFERENCES}$$

These three features collectively along with the static features are used to determine regions of interest with varying resource demands from front end of the pipeline to memory related stalls.

2.2.2 Static Features

While dynamic features provide insights on system resource demands during execution it is important to characterize programs independent of the underlying hardware. Such a static analysis is useful when developing new hardware such as accelerators. We use the vectorization ratio of floating point operations in a program as a static feature and derive this for all the applications using MAQAO-CQA tool.

The vectorization ratio measures how well a region has been vectorized by the compiler. Informally it compares the actual code to an optimally vectorized version of the code supposing that no dependencies exist between instructions. To compute it, first we remove all the instructions that cannot be vectorized on the current ISA such as address computations, branches, or calls. After this step, the instructions left are called *vectorizable instructions*: they are either vector instructions or instructions for which a vector equivalent exist such as load, store or arithmetic operations on floats. The vectorization ratio is defined as the ratio between vector instruction over vectorizable instructions.

MAQAO-CQA (MAQAO Code Quality Analyzer) is the MAQAO module addressing the code quality issues [CRON$^+$14]. Based on a detailed performance model, MAQAO-CQA (i) returns a lower bound on the number of cycles needed to run a binary code fragment, (ii) estimates performance gain if resources were optimally used. It processes the binary code statically, hence the binary code does not have to be executed for analysis and it assumes that most of execution time is spent in loops. MAQAO-CQA compares a binary code against a given machine model and determines the location of the performance bottlenecks. In order to do so, some assumptions are made such as infinite loop trip count and the absence of dynamic hazards such as denormalized numbers and so on. The analysis provide by MAQAO-CQA gives a optimal upper-bound on performance, it is able to accurately detect performance bottlenecks at the micro-processor front-end and arithmetic and logic units levels. For performance bottlenecks caused by memory or cache delays MAQAO-CQA metrics need to be enriched with the previously described dynamic features providing information about the memory latency.

To maintain consistency between the dynamic and static features being used across all the applications, we use the loop identifiers from the MAQAO lprof tool as an input to the MAQAO CQA tool to statically determine the vectorization ratio for the exact same loops that the dynamic features were extracted. This consistency is easy to maintain as both lprof and CQA tools internally use the MAQAO lua plugins.

2.3 Clustering Technique

Clustering is a statistical method to group related data and is used here to determine representative regions of interest from a set of benchmark applications. It groups similar regions into the same cluster such that it suffices to perform simulations for a representative region from a cluster rather than for the entire set of applications, thus saving machine time and man hours.

To cluster the regions of interest, we define similarity using the dynamic and static features discussed above, and use the four metrics CPI, LMR, BMR and Vectorization ratio of floating point operations (Vec. ratio FP). The clustering technique used is similar to the benchmark subsetting methodology [dOCKA$^+$14]. We use these to quantify the similarity between regions and group them into the same cluster. There are multiple ways to compute the similarity using distances between respective vectors. In our approach we use the K-means clustering which uses the Euclidean distance to compute the distance. Automatic methods exist for selecting an optimal K, such as the Elbow method that chooses K so it maximizes the inter-cluster variance over total variance ratio. But in this deliverable, K was empirically selected to a value giving a small number of performance classes easy to work with.

Metrics are not directly comparable. Before computing the distance between two vectors, the vectors must be normalized so each performance metric has the same weight in the distance computation. After normalization the distribution over each metric has zero mean and unit variance.

Euclidean Distance The Euclidean distance between two n-dimensional vectors X and Y is defined as the straight line distance between two points if the two vectors are viewed as single points in a n-dimensional space:

$$distance_{XY} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

2.3.1 Kmeans Clustering

The information presented in this section is taken from wikipedia [Wik].

We use the popular K-means [M+67] clustering approach to partition the regions across all applications and select the regions of interest. K-means clustering aims to partition n observations into k clusters, where each of the n observations belongs to the cluster with the nearest average distance from the cluster centroid, using the Euclidean distance, thus partitioning the data space into Voronoi cells.

Given a set of n observations $(x_1, x_2, \cdots, x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\leq n)$ sets $S_1, S_2, \cdots, S_k$ such that the sum of distances of each point in the cluster to the K center is minimized.

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2$$

where μ_i is the mean of points in S_i.

A common implementation of the k-means clustering algorithm involves iteratively refining the clusters and updating the cluster centroids. Given an initial set of k means $\mu_1, \cdots, \mu_k$, the algorithm alternates between the assignment and update steps [Mac02]. In the assignment step, t, each of the n observations are assigned to the cluster which has the nearest mean,

$$S_i^{(t)} = \{x_p : \|x_p - \mu_i^{(t)}\|^2 \leq \|x_p - \mu_j^{(t)}\|^2 \ \forall j, 1 \leq j \leq k\}$$

where each x_p is assigned to exactly one $S^{(t)}$, even if it can be assigned to two or more of them. The update step recomputes the centroids of the new clusters and updates them as the new means to be used in the next assignment step,

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

This algorithm stops when the clusters no longer change. The outcome of this algorithm depends heavily on the initial values of the cluster centroids. We use a random selection of k observation points among the n points to be the initial cluster centroids and we use the same seed to reproduce the cluster allocation.

2.3.2 Principal Component Analysis

Principal Component Analysis (PCA) is a statistical method used to determine the useful features from a large feature set [Jol]. This method reduces the dimensions of a data set by exploring the correlation between similar variables and converting the set of non-correlated variables into principal components. Each principal component is a linear combination of the original variables. More formally, PCA converts n vectors $X_1, X_2, \cdots, X_n$ into m principal components $Y_1, Y_2, \cdots, Y_m$ such that:

$$Y_i = \sum_{j=1}^{n} a_{ij} X_j, a_{ij} \in \mathbb{R}$$

where a_{ij} is the weight to map the vector X to Y. Such a transformation has two key properties:

$$Var[Y_1] > Var[Y_2] > \cdots Var[Y_m]$$
$$\forall i = j, Cov[Y_i, Y_j] = 0$$

Thus these properties define the transformation such that the first principal component has the largest possible variance. Each succeeding component in turn has the highest variance possible under the constraint that they are not correlated to the previous components. This method reduces the dimensionality of the data while controlling the amount of information that is lost by keeping the components with maximum variance.

3 Selected Regions and Analysis

3.1 Methodology Setup

We used data from 14 programs to apply our methodology, among which 13 programs are from the PARSEC benchmark suite v3.0 and the 14th application is lulesh 2.0. The inputs to the PARSEC benchmark are the native input data set and the lulesh program is executed with input size of 50^3. We measured the data from MAQAO lprof by executing the application on an Intel quad-core processor, i7-4770 CPU operating at 3.40GHz while the MAQAO tool chain is being ported to the ARM architecture. This system has two separate L1 caches of 32KB each for instruction and data respectively. The shared L2 and L3 cache sizes are 256KB and 8MB respectively. All the applications have been compiled with gcc version 4.2.4 with the compilation flags, -O3 -g -funroll-loops and -fprefetch-loop-arrays.

To derive the dynamic features, MAQAO *lprof* tool is used twice. First, the necessary hardware counters to be profiled are given as a separate list in a file using -hwc option and the application profile data represented in MAQAO internal format is collected. Secondly, the profiled information is derived from this data using either -d=SLX or -d=SFX to get loop level or function level data in a comma separated value (csv) format with -ssv=on to additionally derive sample information. In this study, loop level data was always used except for leaf functions that contained no loops for which we fall back on function level data.

The output csv file from *lprof* is parsed to aggregate the dynamic features from different threads and from different region invocation. Only regions that contribute to at least 1% of the total execution time are kept. The system calls are parsed into a separate data sheet. Next, among these application loops that contribute at least 1%, the loop ids as per the lua plugins are parsed and are used as an input to the MAQAO *cqa* tool to derive the static features such as vectorization ratio of floating point operations. The options used with the *cqa* tool are -l to give the list of loop ids, -igp to ignore multiple paths in a loop and -ani to allow analysis of non-innermost loops. Next, the two csv output files from *lprof* and *cqa* are merged together based on the same loopid within an application. Finally the merged dynamic and static feature set per application is again combined to form the complete feature data on which clustering is applied.

We first discuss the hotspots per application reported by lprof after filtering out the regions contributing less than 1% of the total application execution time. Next we discuss the clustering technique applied to all the regions pertaining to a mix of all applications.

When profiling PARSEC and Lulesh to identify the hotspot regions, we always set the number of threads to one. When the benchmark uses a single thread the sum of hotspot's REF_XCLK is always less than 100%. Nevertheless, some PARSEC applications cannot run with a single thread and require additional helper threads. For instance, the bodytrack benchmark requires at least one *Worker* thread and one *Model* thread (cf. `main.cpp@213:mainPthreads`). When the application runs with two or more threads, it is possible for the sum of REF_XCLK values to be larger than 100% because two regions can run concurrently.

3.2 Regions per application

In this section, we report the parsed and filtered lprof output csv files for all the regions within each application, sorted in the increasing order of their execution time. The first column (func_name) represents the function name of the application, the second column (src_info) tells the source line in the source code file name and the third column (REF_XCLK) is the percentage of the time spent by the loop out of the total execution time. If the function name and source line is repeated, it means the compiler has generated multiple binary loops for a single source loop. This can happen in different scenarios such as loop versioning or loop splitting. For some of the function names, the src_info is empty, indicating either that lprof tool is not being able to infer the exact source code line. For each application, the top two loops are described after looking at the individual source codes.

3.2.1 bodytrack

The top most hot-spot in bodytrack application, LoadSet converts the image into binary using the ConvertToBinary function. The second hot-spot FlexDownSample2, down samples image by factor of two with simple anti-aliasing and involves read the entire image and writing half of it back.

Bodytrack is an example of one PARSEC application requiring at least two threads. LoadSet rans in the *Work* thread, whereas FlexDownSample2 rans in the *Model* thread.

func_name	src_info	REF_XCLK
AsyncImageLoader::LoadSet	35,95@AsyncIO.cpp	67.84
FlexDownSample2	84,91@FlexTransform.h	24.20
FlexDownSample2	58,86@FlexTransform.h	24.11
ImageMeasurements::InsideError	46,109@ImageMeasurements.cpp	18.46
ImageMeasurements::InsideError	46,109@ImageMeasurements.cpp	9.22
ImageMeasurements::EdgeError	35,64@ImageMeasurements.cpp	7.03
ImageMeasurements::EdgeError	35,71@ImageMeasurements.cpp	6.68
FlexLoadBMP		5.85
BetaAnnealingFactor	60,533@BinaryImage.h	2.95
TrackingModelPthread::Exec	105,131@TrackingModelPthread.cpp	2.80
TrackingModelPthread::Exec	86,131@TrackingModelPthread.cpp	2.57
TrackingModelPthread::Exec	120,126@TrackingModelPthread.cpp	2.46
FlexLoadBMP	83,235@FlexIO.h	2.34
FlexDownSample2	46,111@FlexTransform.h	1.10

Table 2: Top hot regions for bodytrack

3.2.2 canneal

The top most hot-spot of canneal, create_elem_if_necessary, first parses the database to check if the element exists, else it creates it. The swap_cost function is the second hot-spot of canneal and computes the cost by determining the absolute value of the difference between the current location and the new location.

3.2.3 dedup

The top hot-spot in the dedup kernel is the rabinseg function call that performs a lot of bit-wise boolean arithmetic operations over vectors. The next hot-spot is the sha_block_data_order macro that performs a lot of bit-wise XOR and rotate computations.

3.2.4 facesim

The top hot-spot in the facesim application, namely, Add_Force_Differential function in the DIAGONALIZED_FINITE_VOLUME_3D class performs a significant amount of matrix transpose

func_name	src_info	REF_XCLK
netlist::create_elem_if_necessary	259,2026@netlist.cpp	66.37
netlist_elem::swap_cost	80,533@netlist_elem.cpp	38.33
netlist_elem::swap_cost	89,533@netlist_elem.cpp	27.67
netlist_elem::swap_cost	195,215@netlist_elem.cpp	10.00
netlist_elem::swap_cost	195,215@netlist_elem.cpp	6.70
annealer_thread::Run	195,215@annealer_thread.cpp	5.66
netlist::create_elem_if_necessary	259,2026@netlist.cpp	3.86
netlist::netlist	105,2267@netlist.cpp	3.57
annealer_thread::Run	68,215@annealer_thread.cpp	2.64
netlist_elem::routing_cost_given_loc	56,533@netlist_elem.cpp	2.48
netlist_elem::routing_cost_given_loc	62,533@netlist_elem.cpp	2.46
annealer_thread::Run	195,215@annealer_thread.cpp	1.12

Table 3: Top hot regions for canneal

func_name	src_info	REF_XCLK
rabinseg	87,96@rabin.c	94.89
sha1_block_data_order	239,367@sha_locl.h	91.67
pqdownheap	462,475@trees.c	21.18
deflate_slow	1557,1642@deflate.c	17.66
deflate_slow	1557,1663@deflate.c	10.82
TreeFind	29,34@tree.c	6.13
longest_match	1027,1164@deflate.c	3.78
compress_block	1084,1114@trees.c	2.74
build_tree	669,690@trees.c	1.96
copy_block	1216,1217@trees.c	1.79
gen_bitlen	514,528@trees.c	1.62
rabinseg	72,85@rabin.c	1.22
DeleteMin	85,94@binheap.c	1.11

Table 4: Top hot regions for dedup

computations and vector arithmetic. The next hot-spot is the Update_Position_Based_State function call in the DIAGONALIZED_FINITE_VOLUME_3D class and performs singular value decomposition and matrix determinant computation.

func_name	src_info	REF_XCLK
PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	89,1096@DIAGONALIZED_FINITE_VOLUME_3D.cpp	25.96
PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	24,696@DIAGONALIZED_FINITE_VOLUME_3D.cpp	9.36
PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	30,617@DIAGONALIZED_FINITE_VOLUME_3D.cpp	7.48
PhysBAM::DEFORMABLE_OBJECT	24,377@DEFORMABLE_OBJECT.cpp	2.10
PhysBAM::DIAGONALIZED_FACE_3D	129,629@DIAGONALIZED_FACE_3D.h	1.61
PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	89,1094@DIAGONALIZED_FINITE_VOLUME_3D.cpp	1.57
PhysBAM::COLLISION_PENALTY_FORCES	18,870@locale_facets.h	1.10

Table 5: Top hot regions for dedup

3.2.5 ferret

The top most hot-spot of the ferret application is the image_extract_helper code that computes the features for every image by first extracting boxes using the function call box_set_insert_pxl, followed by extracting colours and region size using map.rgn_sz and finally assigns weights. The second hot-spot is the image_segment function call that is very memory intensive as it first allocates each pixel as a region, and computes the maximum value among the red, green and blue values with each of the neighbours, and finally merges similar neighbours into smaller regions and assigns the mean colour.

func_name	src_info	REF_XCLK
image_extract_helper	261,323@extract.c	23.39
image_segment	402,439@srm.c	19.36
isOptimal	419,423@emd.c	17.13
find_set	122,122@srm.c	14.77
dist_L2_float		14.59
findBasicVariables	347,356@emd.c	14.23
russel	695,699@emd.c	13.98
image_extract_helper	298,305@extract.c	12.84
findBasicVariables	377,386@emd.c	12.65
LSH_query_bootstrap	217,257@LSH_query.c	11.16
vertical	149,155@image.c	8.41
ckh_alloc_table	142,144@cuckoo_hash.c	7.69
horizontal	60,102@image.c	7.36
horizontal	60,106@image.c	7.26
image_extract_helper	282,284@extract.c	6.42
vertical	144,159@image.c	6.41
LSH_query_bootstrap	257,257@LSH_query.c	5.73
isOptimal	418,423@emd.c	5.15
ycc_rgb_convert	144,153@jdcolor.c	3.36
bucket_sort	157,189@srm.c	3.23
image_segment	246,353@srm.c	2.96
decode_mcu	1059,1078@jdhuff.c	2.73
findLoop	545,607@emd.c	2.55
findBasicVariables	372,394@emd.c	2.47
image_segment	246,471@srm.c	2.43
findBasicVariables	342,364@emd.c	2.35
image_segment	485,494@srm.c	2.12
russel	690,699@emd.c	2.07
LSH_query_bootstrap	217,257@LSH_query.c	2.07
jpeg_idct_16x16	2561,2805@jidctint.c	2.00
jpeg_idct_islow	171,408@jidctint.c	1.79
dist_L2_float		1.54
emdinit	214,218@emd.c	1.21
image_segment	246,321@srm.c	1.15
cass_result_merge_lists	284,284@util.c	1.11

Table 6: Top hot regions for dedup

3.2.6 fluidanimate

The ComputeForcesMT function, is a compute intensive program involving square root, division and multiplication of 3x3x3 vectors and hence takes up 50% of the execution time. The next hot-spot ComputeDensitiesMT function also operates on 3x3x3 vectors but only does comparisons and addition operations and some modulo-arithmetic to compute remainders.

func_name	src_info	REF_XCLK
ComputeForcesMT	214,853@pthreads.cpp	44.79
ComputeDensitiesMT	341,751@pthreads.cpp	23.32
ComputeDensitiesMT	341,751@pthreads.cpp	15.42
ComputeForcesMT	214,853@pthreads.cpp	5.30
InitSim	229,254@pthreads.cpp	4.17
InitSim	48,383@pthreads.cpp	4.17
RebuildGridMT	555,629@pthreads.cpp	2.26
SaveFile	48,455@pthreads.cpp	2.08
AdvanceParticlesMT	346,1111@pthreads.cpp	1.47

Table 7: Top hot regions for fluidanimate

3.2.7 freqmine

The top most hot-spot of freqmine application is the FPArray_scan2_DB and involves parsing the entire data base stored in the form of a tree to determine the frequency count of an item. The second hot-spot function, FPArray_conditional_pattern_base traverses the FP tree to count the patters formed by traversing the frequencies of individual items.

func_name	src_info	REF_XCLK
FPArray_scan2_DB	361,369@fp_tree.cpp	16.41
FPArray_conditional_pattern_base	309,310@fp_tree.cpp	9.07
FP_tree::insert	949,966@fp_tree.cpp	8.49
FPArray_scan2_DB	350,381@fp_tree.cpp	7.02
transform_FPTree_into_FPArray	105,172@fp_tree.cpp	4.93
FPArray_conditional_pattern_base	301,312@fp_tree.cpp	4.61
FP_tree::fill_count	1035,1039@fp_tree.cpp	3.77
transform_FPTree_into_FPArray	155,166@fp_tree.cpp	3.11
FP_tree::FP_growth	1241,1525@fp_tree.cpp	2.00

Table 8: Top hot regions for freqmine

3.2.8 raytrace

The topmost hot-spot in the raytrace application is the tracer using bounding volume hierarchy (BVH) and provides Axis Aligned Bounding Boxes (AABB) in space to infer whether the ray should trace that volume in space or not. The TraverseBVH computes the different signs of the ray directions involving lot of vector shuffle operations. The TraverseBVH calls the RayPacketIntersectAABB function which determines if the ray packet intersects the AABB and involves many vector comparison operations to compute min and max.

func_name	src_info	REF_XCLK
RTTL::TraverseBVH	10,784@BVH.hxx	30.33
RTTL::TraverseBVH	53,784@BVH.hxx	11.97
Context::renderFrame	66,702@render.cxx	8.52
RTTL::TraverseBVH	53,567@BVH.hxx	7.97
RTTL::TraverseBVH	10,784@BVH.hxx	5.99
std::map¡std::pair	154,985@stl_map.h	5.00
std::map¡std::pair	154,985@stl_tree.h	4.61
Context::renderFrame	78,703@render.cxx	3.44
Context::renderFrame	183,616@render.cxx	3.37
RTTL::BinnedAllDimsSaveSpace	46,784@BinnedAllDimsSaveSpace.cxx	2.72

Table 9: Top hot regions for raytrace

3.2.9 streamcluster

The clustering algorithm of streamcluster iteratively computes the cost until an improvement less than a threshold value is reached. The pFL is the main function which in turn calls the pgain function which computes the cost using the Euclidean distance and then assigns points to centers.

func_name	src_info	REF_XCLK
pFL	653,1207@streamcluster.cpp	100.00
streamCluster	1638,1641@streamcluster.cpp	23.11
SimStream::read	1763,1767@streamcluster.cpp	14.62
streamCluster	1633,1643@streamcluster.cpp	6.60
pFL	652,1207@streamcluster.cpp	6.41
pFL	652,653@streamcluster.cpp	5.49
pspeedy	653,703@streamcluster.cpp	1.67

Table 10: Top hot regions for streamcluster

3.2.10 swaptions

The top hot-spot function in swaptions applications involves calculating the Hamilton-Jacobi-Bellman (HJB) path for a given input set of stock maturities. It first sequentially generates random number which is 10% of the execution time and then generates the HJM paths using the stochastic factors as inputs. The second hot-spot is the free_dmatrix function which is used to free memory for a two-dimensional vector.

func_name	src_info	REF_XCLK
HJM_SimPath_Forward_Blocking	73,154@HJM_SimPath_Forward_Blocking.cpp	17.03
HJM_SimPath_Forward_Blocking	73,162@HJM_SimPath_Forward_Blocking.cpp	3.80
Discount_Factors_Blocking	392,395@HJM.cpp	3.60
HJM_SimPath_Forward_Blocking	73,121@HJM_SimPath_Forward_Blocking.cpp	2.16
HJM_SimPath_Forward_Blocking	66,67@HJM_SimPath_Forward_Blocking.cpp	1.06

Table 11: Top hot regions for swaptions

3.2.11 vips

The image processing application vips has the top hot-spot as lintran_gen function. This consists of a switch case statement based on which a loop that performs vector arithmetic (multiplication and addition) is called. The switch case is to resolve the different input and output data types of vectors, among char, float, double and so on. The second hot-spot is imb_XYZ2Lab which does image quantization using vectors and also performs division operation.

func_name	src_info	REF_XCLK
lintran_gen	210,226@im_lintra.c	12.84
imb_XYZ2Lab	104,141@im_XYZ2Lab.c	11.07
recomb_buf	73,87@im_recomb.c	8.6
imb_Lab2XYZ	66,101@im_Lab2XYZ.c	7.59
extract_band	93,125@im_extract.c	4.8
conv_gen	344,344@im_convsep.c	4.73
conv_gen	344,344@im_convsep.c	4.71
vips_threadpool_run	847,863@threadpool.c	4.65
extract_band	93,125@im_extract.c	4.57
imb_Lab2LabQ	88,126@im_Lab2LabQ.c	4.24
imb_Lab2LabQ	88,126@im_Lab2LabQ.c	4.01
conv_gen	344,344@im_convsep.c	3.38
affinei_gen	184,323@im_affine.c	3.26
imb_LabS2LabQ	68,116@im_LabS2LabQ.c	2.61
conv_gen	344,344@im_convsep.c	2.45
imb_LabQ2Lab	68,99@im_LabQ2Lab.c	2.40
imb_LabQ2Lab	68,99@im_LabQ2Lab.c	2.33
join_bands	109,147@im_gbandjoin.c	2.33
imb_Lab2XYZ	66,101@im_Lab2XYZ.c	2.33
imb_LabQ2LabS	63,85@im_LabQ2LabS.c	2.10
join_bands	109,147@im_gbandjoin.c	2.05
imb_LabQ2disp	84,116@im_LabQ2disp.c	1.26
lintra1_gen	145,165@im_lintra.c	1.23

Table 12: Top hot regions for vips

3.2.12 x264

The first hot-spot is the block_residual_cabac code for the H.264 encoder and is a Context Adaptive Binary Arithmetic Coder (CABAC). This function calls the cabac_encode_decision function which is implemented in assembly and does boolean arithmetic operations for vectors. The next hot-spot in x264 application is the pixel averaging code using SIMD extension and is also implemented in x86 assembly using the SSE instruction set and is a vectorized code.

x264 has a very flat profile. Performance is distributed among many kernels and there are no large hotspot loops or leaf functions. This kind of application is not the best fit for the proposed methodology which focuses on large loops.

3.2.13 lulesh

The top-most function that takes 18% of the execution time is computing the hourglass modes. The hour glass modes computation involves vector addition and multiplication. The imple-

func_name	src_info	REF_XCLK
block_residual_write_cabac		3.79
block_residual_write_cabac		3.72
x264_pixel_avg2_w16_sse2		2.25
x264_mc_chroma_ssse3		1.74
x264_me_search_ref		1.46
x264_mb_analyse_intra		1.24
block_residual_write_cabac		1.22
block_residual_write_cabac		1.21
x264_mb_analyse_intra		1.09
x264_hpel_filter_ssse3		0.97

Table 13: Top hot regions for x264

mentation uses the Flanagan-Belytschko kinematic hourglass filter which is used commonly in Lagrange finite element hydrocodes [HKG11]. The second hot-spot is the integrating function that integrate the volumetric stress contributions for each element. This involves first collecting the node coordinates into local arrays and then computing the normal vectors, involving vector multiplication and addition.

func_name	src_info	REF_XCLK
_ZL28CalcFBHourglassForceForElemsR6DomainPdS1	50,991@lulesh.cc	18.20
_ZL23IntegrateStressForElemsR6DomainPdS1	270,611@lulesh.cc	10.79
_Z22CalcKinematicsForElemsR6DomainPddi	46,1594@lulesh.cc	10.68
_ZL28CalcHourglassControlForElemsR6DomainPdd	270,1062@lulesh.cc	9.62
_ZL31CalcMonotonicQGradientsForElemsR6DomainPd	46,1783@lulesh.cc	6.58
_ZL18CalcEnergyForElemsPdS	46,2126@lulesh.cc	3.94
_ZL15EvalEOSForElemsR6DomainPdiPii	611,2279@lulesh.cc	3.68
_ZL18CalcEnergyForElemsPdS	46,2175@lulesh.cc	3.62
_ZL28CalcMonotonicQRegionForElemsR6DomainiPdd	290,1947@lulesh.cc	3.23
_ZL20CalcPressureForElemsPdS	54,2071@lulesh.cc	3.12
_ZL15EvalEOSForElemsR6DomainPdiPii	2286,2288@lulesh.cc	2.86
_ZL18CalcEnergyForElemsPdS	46,2198@lulesh.cc	2.54
_ZL20CalcPressureForElemsPdS	2054,2055@lulesh.cc	1.83
_ZL18CalcEnergyForElemsPdS	54,2138@lulesh.cc	1.13

Table 14: Top hot regions for lulesh

3.3 PCA and Clustering

PCA technique determines correlation between the features in a data set and remove strongly correlated features, thus reducing the dimensionality of the data. We applied the PCA to the four features used in our methodology, CPI, BMR, LMR and Vec. Ratio FP. Table 15 shows the correlation among these features. As indicated by the values in the table, they are very close to zero and hence the feature vectors chosen are not correlated to each other. Thus we perform clustering using all of these four features.

We use R [R C13] to perform k-means clustering on the four-dimensional feature set of all regions extracted from the 14 programs. Figure 2 plots the 243 regions clustered into five

	CPI	BMR	Vec. ratio FP	LMR
CPI	1.00	-0.02	-0.10	0.22
BMR	-0.02	1.00	-0.14	0.01
Vec. ratio FP	-0.10	-0.14	1.00	-0.17
LMR	0.22	0.01	-0.17	1.00

Table 15: Correlation among features

clusters along the two principal component axes. The direction of the four feature vectors in 2-D space along the two principal components is also shown. As is seen from this plot, the five clusters are along the three feature vectors. Next we discuss the regions of interest within each cluster.

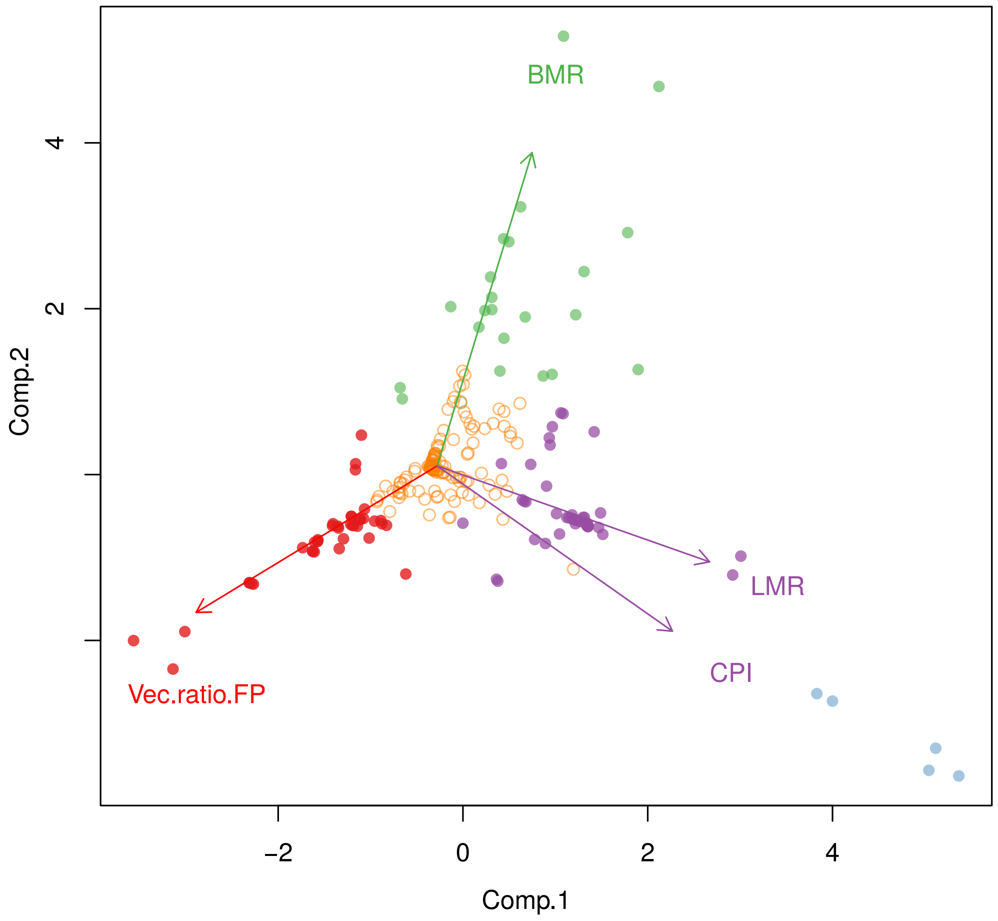

Figure 2: PCA analysis

Application	FunctionName	srcinfo	REF_XCLK	CPI	BMR	Vec. ratio (FP)	LMR
x264	X264:x264_mc_chroma_ssse3		341	0.9190557806	0.0003436426	100	0.2496688742
x264	X264:x264_hpel_filter_ssse3		191	0.3170313369	0.008387698	100	0
lulesh2.0	LL:CalcPressureForElemsPd	2054,2055@lulesh.cc	1.83 (768)	0.7621006183	5.90772139186E-005	85.7142857143	0.0532241556
vips	VI:lintran_gen	210,226@im_lintra.c	551	0.2850407061	2.99742221689E-005	61.5384615385	0.0423728814
vips	VI:lintran_gen	210,226@im_lintra.c	533	0.2837128697	0	61.5384615385	0.0263157895
vips	VI:lintran_gen	210,226@im_lintra.c	497	0.2844059044	5.96961466137E-005	61.5384615385	0.025862069
vips	VI:lintran_gen	210,226@im_lintra.c	489	0.2843199557	8.98903337928E-005	61.5384615385	0.0526315789
vips	VI:lintra1_gen	145,165@im_lintra.c	53	0.4085622371	0	42.8571428571	0
vips	VI:lintra1_gen	145,165@im_lintra.c	52	0.4093173891	0	42.8571428571	0.0769230769
vips	VI:lintra1_gen	145,165@im_lintra.c	50	0.3935045317	0	42.8571428571	0.0666666667
vips	VI:lintra1_gen	145,165@im_lintra.c	46	0.4186807654	0	42.8571428571	0.0666666667
lulesh2.0	LL:CalcFBHourglassForceForElemsR6DomainPd.6	50,991@lulesh.cc	18.20 (7629)	0.4402804312	0.0007688544	39.696969697	0.6160849772
vips	VI:imb_Lab2XYZ	66,101@im_Lab2XYZ.c	326	0.8236700077	0.0028558373	39.3442622951	0
vips	VI:imb_Lab2XYZ	66,101@im_Lab2XYZ.c	322	0.832627608	0.0017020311	39.3442622951	0
vips	VI:imb_Lab2XYZ	66,101@im_Lab2XYZ.c	302	0.828379526	0.0019522278	39.3442622951	0.1428571429
vips	VI:imb_Lab2XYZ	66,101@im_Lab2XYZ.c	299	0.8192891068	0.0018150879	39.3442622951	0
vips	VI:imb_Lab2XYZ	66,101@im_Lab2XYZ.c	1	0.568627451	0	39.3442622951	0
ferret	FT:isOptimal	419,423@emd.c	9178	0.5166726157	0.048057535	33.3333333333	0.0196078431
ferret	FT:russel	695,699@emd.c	7490	0.6865351688	0.0444591904	33.3333333333	0.0183486239
ferret	FT:isOptimal	418,423@emd.c	2762	0.5585140904	0.0676456573	33.3333333333	0
vips	VI:imb_Lab2LabQ	88,126@im_Lab2LabQ.c	182	0.3934977578	0.0029943855	33.3333333333	0.04
vips	VI:imb_Lab2LabQ	88,126@im_Lab2LabQ.c	179	0.3925960082	0.003115653	33.3333333333	0
vips	VI:imb_Lab2LabQ	88,126@im_Lab2LabQ.c	177	0.3898070718	0.0039810898	33.3333333333	0.0344827586
vips	VI:imb_Lab2LabQ	88,126@im_Lab2LabQ.c	172	0.3927655377	0.0044676098	33.3333333333	0
ferret	FT:t_out	387,393@ferret-pthreads.c	0	1.8333333333	0	33.3333333333	0
vips	VI:imb_XYZ2Lab	104,141@im_XYZ2Lab.c	475	0.7312719733	0.0009567089	29.0909090909	0.0224719101

Table 16: High vectorization ratio regions

Application	FunctionName	srcinfo	REF_XCLK	CPI	BMR	Vec. ratio (FP)	LMR
canneal	CA:annealer_thread::Run	195,215@annealer_thread.cpp	1234	30.252245509	0	0	0.9644902635
canneal	CA:netlist_elem::routing_cost_given_loc	56,533@netlist_elem.cpp	179	41.8503401361	0	0	0.9460784314
canneal	CA:netlist_elem::routing_cost_given_loc	62,533@netlist_elem.cpp	177	43.8111888112	0.019379845	0	0.8254716981
canneal	CA:netlist_elem::swap_cost_t	89,533@netlist_elem.cpp	6035	47.464974142	0.0139671649	0	0.7870351555
canneal	CA:netlist_elem::swap_cost_t	80,533@netlist_elem.cpp	8362	32.7072727273	0.0059490085	0	0.7054772056

Table 17: Regions for big processors:memory-bound

3.4 Regions of Interest

3.4.1 Accelerators: High vectorization ratio

Among the four features, the floating point vectorization feature classifies interesting regions within applications that will benefit from the use of accelerators. Within these regions, regions that have a high floating point vectorization ratio classify as more suitable for accelerators because a high vectorization ratio indicates the code benefits from the use of a specialized vector processing unit and is limited by the current computational capabilities of the general processor. Such regions identified by applying our methodology are shown in Figure 3 and Table 16. The regions here belong to x264 application, lulesh and the vips program. These programs do image processing or intensive numerical computation which are usually present good opportunities for vectorization. Thus these regions can be used by the architectural simulation team to perform simulations of accelerators.

3.4.2 Regions for executing on big processor: memory bound

As indicated in Table 17, this cluster consists of regions with high CPI, and high LMR. The high CPI indicates that these regions are not optimal either due to the style of code or due to the resource demands from processing. The regions with high LMR that have extremely large CPI, especially from canneal application, are regions that can be optimized for the big processor by increasing the memory bandwidth.

The regions of this cluster are shown in Figure 4. As indicated the high CPI and high LMR regions are more suitable for big processor and there is a need to optimize the memory

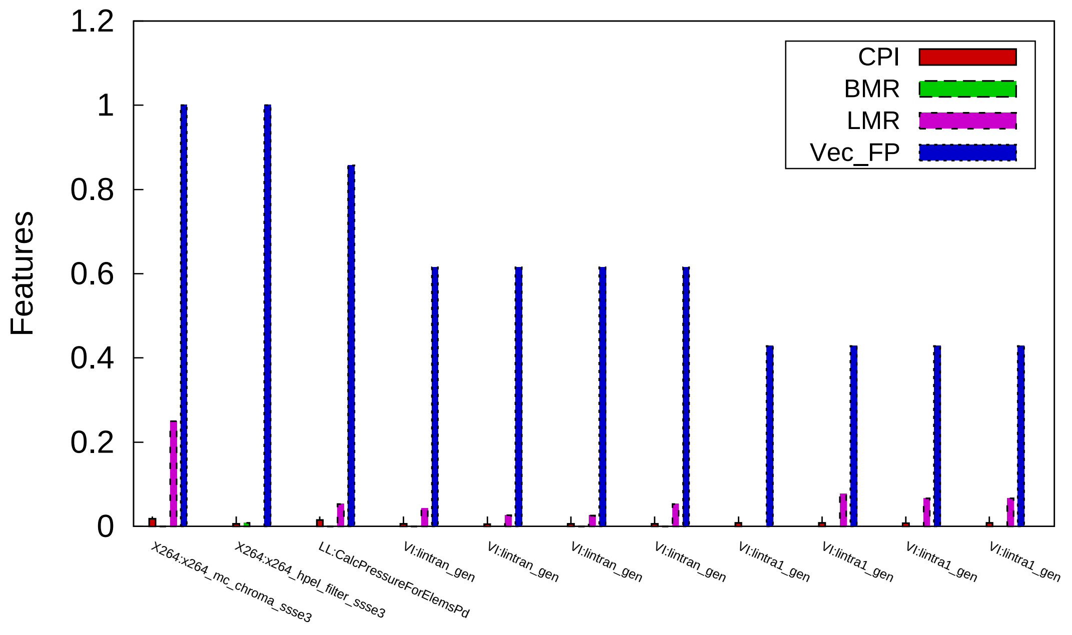

Figure 3: High vectorization ratio regions

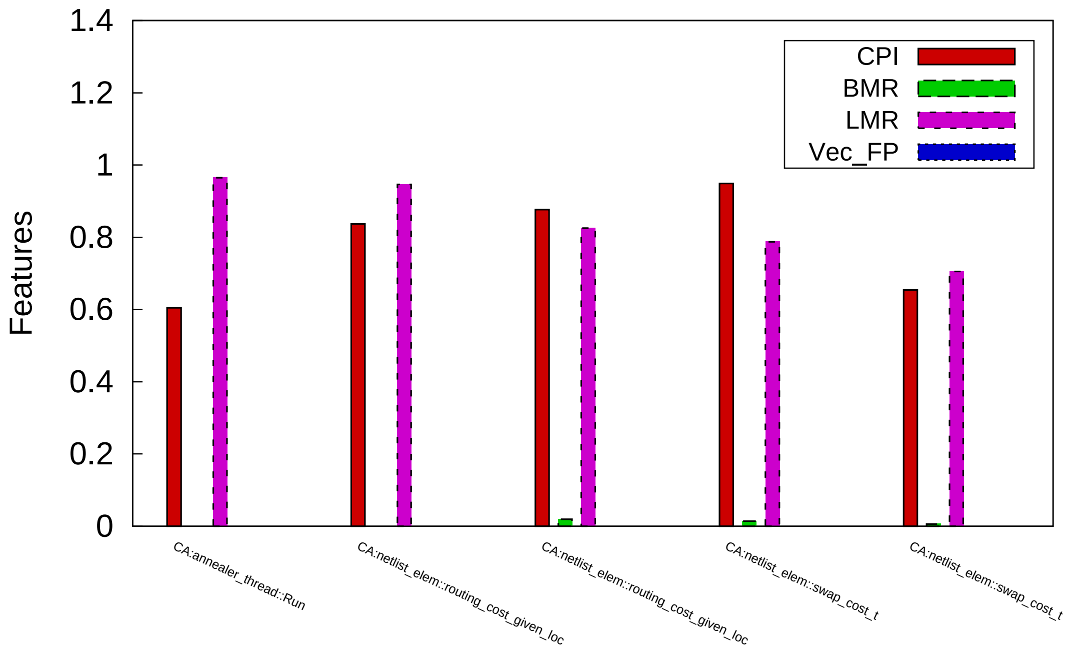

Figure 4: Regions for big processors:memory-bound

Application	FunctionName	srcinfo	REF_XCLK	CPI	BMR	Vec. ratio (FP)	LMR
x264	X264:x264_mb_analyse_intra		243	0.5878063011	0.3057658707	0	0
rtview	RT:RTTL::TraverseBVH	10,784@BVH.hxx	363	0.7363057325	0.2955558973	0	0.652173913
rtview	RT:RTTL::TraverseBVH	10,784@BVH.hxx	2233	1.9745778007	0.1977838494	0	0.6512488437
dedup	DD:build_tree	669,690@trees.c	68	1.4061281337	0.188952381	0	0
ferret	FT:image_segment	485,494@srm.c	48	1.2847619048	0.1638168781	0	0
x264	X264:block_residual_write_cabac		745	0.6863478039	0.1633888048	0	0
freqmine	FM:FP_tree::fill_count	1035,1039@fp_tree.cpp	3074	0.8868042849	0.1610130031	0	0.5277777778
rtview	RT:RTTL::TraverseBVH	53,784@BVH.hxx	4459	0.5860455114	0.1454337053	7.7586206897	0.6882933709
ferret	FT:russel	654,661@emd.c	517	1.5008998691	0.1406214039	16.6666666667	0
x264	X264:block_residual_write_cabac		732	0.5420502586	0.1357829524	0	0
canneal	CA:netlist::create_elem_if_necessary	259,2026@netlist.cpp	4784	10.193207922	0.1287188828	0	0.4504034761
ferret	FT:image_segment	246,471@srm.c	55	1.2087912088	0.1274883524	5.8823529412	0.1111111111
ferret	FT:findBasicVariables	372,394@emd.c	1322	1.2472714689	0.1244253427	0	0
canneal	CA:netlist::create_elem_if_necessary	259,2026@netlist.cpp	278	3.6967769296	0.1242019733	0	0.0833333333
dedup	DD:scan_tree	723,740@trees.c	32	0.9657258065	0.1139364303	0	0
streamcluster	SC:pFL	652,1207@streamcluster.cpp	697	0.7956336966	0.1123361144	13.0434782609	0.71875
dedup	DD:pqdownheap	462,475@trees.c	734	0.8729954181	0.1017937075	0	0
freqmine	FM:transform_FPTree_into_FPArray	105,172@fp_tree.cpp	4019	0.6430468816	0.0999479342	0	0.1838006231
canneal	CA:annealer_thread::Run	68,215@annealer_thread.cpp	575	1.7088576363	0.0975449652	25	0.3433962264
ferret	FT:findBasicVariables	347,356@emd.c	7625	0.9413249097	0.0919153111	25	0
ferret	FT:jpeg_idct_islow	194,290@jidctint.c	9	0.4208289054	0.0843373494	0	0.5
freqmine	FM:FPArray_conditional_pattern_base	301,312@fp_tree.cpp	3754	0.526147384	0.0779483231	0	0.2259887006

Table 18: Region for executing on big processor: branch-bound

imbalance as pointed out by these regions.

3.4.3 Region for executing on big processor: branch bound

The third cluster consists of regions that have a high branch misprediction ratio compared to all the other clusters. While the other three clusters have BMR less than 0.1, there are many regions within this cluster with BMR greater than 0.1. Hence such regions are more suitable for the big processor as it implements complex out-of-order execution pipelines and hardware branch prediction engines. Thus the top regions from this cluster as shown in the below table can be used for micro-architecture optimizations and simulations to improve the performance of the big processor in the big.LITTLE heterogeneous system.

While Table 18 lists the clusters by sorting the BMR, the top ten regions are plotted in Figure 5.

3.4.4 Regions for little processor

Regions that have a high LMR but do not impact the CPI of the processor perform efficiently. Thus these regions can be used to save energy by executing on the little processor. Regions in this group can be executed on the little processor and save energy as the CPI is very low anyway.

While the second region had high LMR along with a high CPI, this cluster of regions have high LMR with low CPI as shown in Figure 6 and Table 6. Thus, these regions can be used for simulation of efficient memory request processing within a CPU.

3.4.5 Balanced Cluster

The central cluster is more suitable for low overhead processing as this consists of regions in the center of the PCA clustering plot shown in Figure 2 and listed in Table 20. These regions do not have a high value in any of the four features, namely, CPI, LMR, BMR and floating point vectorization ratio. Thus these regions represent balanced code in the programs and can be used to simulate new architecture features and ensure that the new features have not disturbed the balanced code.

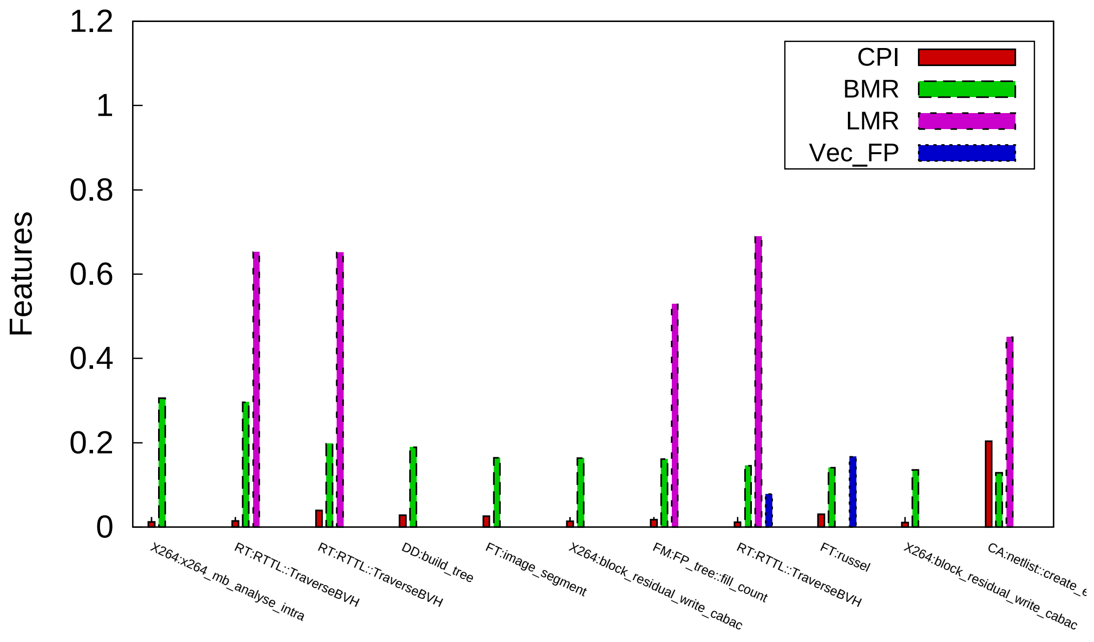

Figure 5: Region for executing on big processor: branch-bound

Application	FunctionName	srcinfo	REF_XCLK	CPI	BMR	Vec. ratio (FP)	LMR
dedup	DD:TreeFind	29,34@tree.c	22	2.7049180328	0.0013157895	0	1
ferret	FT:horizontal	60,106@image.c	69	0.4741264082	0.0020214586	12.8205128205	1
ferret	FT:image_extract_helper	282,284@extract.c	14	0.435499515	0.0036452005	0	1
bodytrack	BT:AsyncImageLoader::LoadSet	35,95@AsyncIO.cpp	116	0.3092598376	0.0030959752	0	1
ferret	FT:jpeg_idct_16x16	2561,2805@jidctint.c	19	0.2973202892	0	0	1
streamcluster	SC:pspeedy	653,703@streamcluster.cpp	182	1.238018849	0	0	0.9889867841
streamcluster	SC:pFL	653,1207@streamcluster.cpp	9753	1.2738193169	0.0006416534	0	0.9865577889
streamcluster	SC:pFL	653,1207@streamcluster.cpp	12486	1.2716008448	0.0005942845	0	0.9822635135
streamcluster	SC:pFL	653,1207@streamcluster.cpp	14007	1.2722553763	0.0006505123	0	0.9818769849
streamcluster	SC:pFL	653,1207@streamcluster.cpp	12459	1.2697582655	0.0003634759	0	0.981864864
streamcluster	SC:pFL	653,1207@streamcluster.cpp	12594	1.2694558073	0.000520156	0	0.978792294
streamcluster	SC:pspeedy	653,703@streamcluster.cpp	180	1.2594473791	0.002020202	0	0.9782135076
streamcluster	SC:pspeedy	653,703@streamcluster.cpp	193	1.24	0.0018382353	0	0.9754601227
facesim	FS:PhysBAM::DIAGONALIZED_FACE_3D	129,629@DIAGONALIZED_FACE_3D.h	969	0.6573213499	0.0014644351	7.3333333333	0.9751552795
streamcluster	SC:pFL	652,653@streamcluster.cpp	762	0.5229265896	0.0001711157	0	0.9705329154
streamcluster	SC:pFL	652,653@streamcluster.cpp	604	0.5158136721	0	0	0.947284345
streamcluster	SC:pFL	652,653@streamcluster.cpp	824	0.5146339399	6.24804748516E-005	0	0.9322228604
streamcluster	SC:pFL	652,653@streamcluster.cpp	799	0.5193271632	6.86553842985E-005	0	0.9278350515
facesim	FS:PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	30,617@DIAGONALIZED_FINITE_VOLUME_3D.cpp	4511	0.3457450976	0.0002176173	11.9047619048	0.9227272727
ferret	FT:LSH_query_bootstrap	217,257@LSH_query.c	231	0.4423296725	0.060362173	0	0.9039548023
streamcluster	SC:pFL	652,653@streamcluster.cpp	789	0.5190960751	3.44613688056E-005	0	0.9032059186
streamcluster	SC:streamCluster	1633,1643@streamcluster.cpp	14	2.3101604278	0	28.5714285714	0.9
canneal	CA:netlist_elem::swap_cost_t	195,215@netlist_elem.cpp	1462	0.9595214282	0.0034115994	0	0.8894577171
dedup	DD:rabinseg	87,96@rabin.c	390	0.6449512995	0	0	0.8823529412
canneal	CA:netlist_elem::swap_cost_t	195,215@netlist_elem.cpp	2181	1.7105886102	0	0	0.8784828592
streamcluster	SC:pFL	652,1207@streamcluster.cpp	938	0.796890914	0.0897373541	13.0434782609	0.8552437223
streamcluster	SC:pFL	652,1207@streamcluster.cpp	948	0.789859617	0.0897717296	13.0434782609	0.8409415121
streamcluster	SC:pFL	652,1207@streamcluster.cpp	901	0.7732187784	0.0711707065	13.0434782609	0.8173652695
ferret	FT:dist_L2_float@0x424790		1629	0.6177589656	0.000031375	0	0.8153590898
streamcluster	SC:pFL	652,1207@streamcluster.cpp	842	0.7876497548	0.0789865872	13.0434782609	0.8148984199
canneal	CA:netlist::netlist	105,2267@netlist.cpp	257	5.5481727575	0.0199700449	0	0.7716049383
fluidanimate	FA:AdvanceParticlesMT	346,1111@pthreads.cpp	773	1.0676538677	0.058685446	8.3333333333	0.7480848556
rtview	RT:RTTL::TraverseBVH	53,567@BVH.hxx	2971	0.340815553	0.0147892484	0	0.7232704403
vips	VI:imb_LabQ2disp	84,116@im_LabQ2disp.c	45	0.28290138	0	0	0.6315789474
vips	VI:imb_LabQ2disp	84,116@im_LabQ2disp.c	54	0.2862836267	0	0	0.619047619
ferret	FT:LSH_query_bootstrap	257,257@LSH_query.c	640	0.3189199314	0.0005933147	0	0.6040609137
facesim	FS:PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	24,696@DIAGONALIZED_FINITE_VOLUME_3D.cpp	5648	0.6982649098	0.0415843228	11.8483412322	0.5646502836
fluidanimate	FA:RebuildGridMT	555,629@pthreads.cpp	1187	0.8471886495	0.0260629304	0	0.5580469405
vips	VI:imb_LabQ2disp	84,116@im_LabQ2disp.c	37	0.2783729494	0.0024449878	0	0.5
freqmine	FM:transform_FPTree_into_FPArray	155,166@fp_tree.cpp	2534	0.4815067513	0.0358818723	0	0.4614886731
lulesh2.0	LL:CalcHourglassControlForElemsR6DomainPdd	270,1062@lulesh.cc	9.62 (4033)	0.4432769158	0.0028604119	2.5039123631	0.460627895
fluidanimate	FA:ComputeDensitiesMT	341,751@pthreads.cpp	8098	0.4877603215	0.0467830841	3.125	0.4470149254
fluidanimate	FA:ComputeDensitiesMT	341,751@pthreads.cpp	12249	0.5795381102	0.0677742219	3.125	0.4463190184
ferret	FT:image_extract_helper	298,305@extract.c	28	0.3734817814	0.008605852	0	0.4444444444

Table 19: Regions for little processor

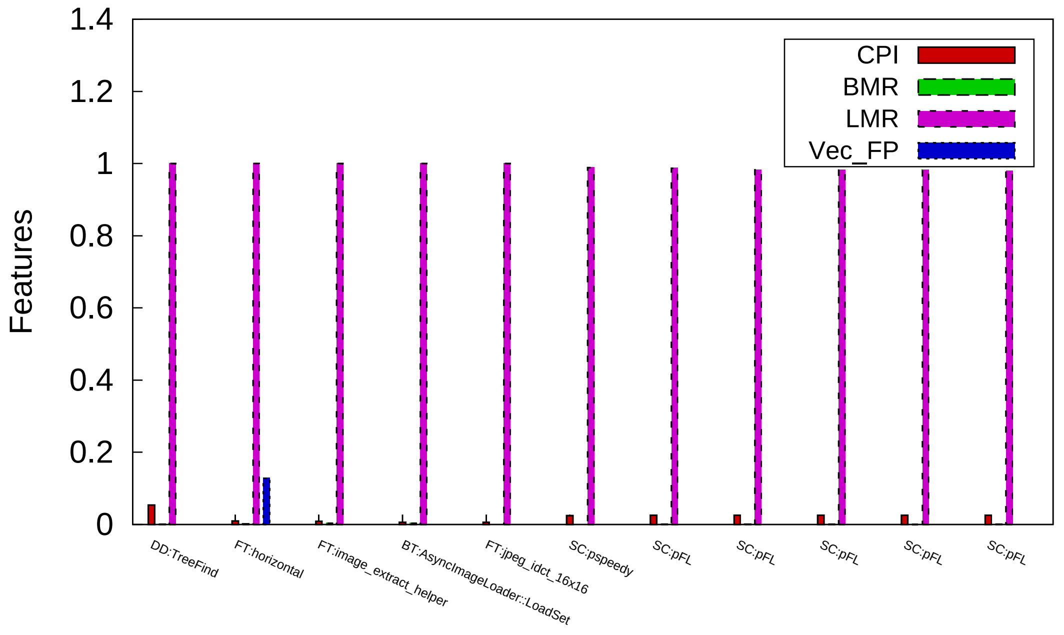

Figure 6: Regions for little processor

Application	FunctionName	srcinfo	REF_XCLK	CPI	BMR	Vec. ratio (FP)	LMR
lulesh2.0	LL:CalcHourglassControlForElemsR6DomainPdd	270,1062@lulesh.cc	4033	0.4432769158	0.0028604119	2.5039123631	0.460627895
lulesh2.0	LL:CalcMonotonicQGradientsForElemsR6DomainPd	46,1783@lulesh.cc	2757	0.5500650347	0.0047250859	6.7885117493	0.3174767322
lulesh2.0	LL:CalcEnergyForElemsPd	46,2126@lulesh.cc	1652	0.4719294626	0.007744916	10	0.0076481836
lulesh2.0	LL:EvalEOSForElemsR6DomainPdiPii	611,2279@lulesh.cc	1542	1.0130426319	0.0002629503	0	0.1072453977
lulesh2.0	LL:CalcEnergyForElemsPd	46,2175@lulesh.cc	1519	0.3944571938	0.0045034834	9.5238095238	0.0013351135
lulesh2.0	LL:CalcMonotonicQRegionForElemsR6DomainiPdd	290,1947@lulesh.cc	1353	0.5359928812	0.0020440974	15.3110047847	0.4151421334
lulesh2.0	LL:CalcPressureForElemsPdS_S_S_S_S_dddiPi	54,2071@lulesh.cc	1306	0.3303691336	0.0033421759	0	0.0045857536
lulesh2.0	LL:CalcEnergyForElemsPd	46,2198@lulesh.cc	1065	0.4300013526	0.0095311754	16.6666666667	0
lulesh2.0	LL:IntegrateStressForElemsR6DomainPd	270,611@lulesh.cc	4521	0.4512985477	0.0035608309	10.3144654088	0.3444444444
fluidanimate	FA:ComputeForcesMT	214,853@pthreads.cpp	23528	0.4900858963	0.0469587393	4.958677686	0.3136682243
facesim	FS:PhysBAM::DIAGONALIZED_FINITE_VOLUME_3D	89,1096@DIAGONALIZED_FINITE_VOLUME_3D.cpp	15654	0.3279776605	0.0006733533	9.5238095238	0.2273301194
freqmine	FM:FPArray_scan2_DB	361,369@fp_tree.cpp	13374	0.4587669591	0.0344150449	0	0.1665043817
fluidanimate	FA:ComputeDensitiesMT	341,751@pthreads.cpp	12249	0.5795381102	0.0677742219	3.125	0.4463190184
rtview	RT:RTTL::TraverseBVH	10,784@BVH.hxx	11301	0.4742128752	0.0621480709	4.7008547009	0.3613138686
fluidanimate	FA:ComputeDensitiesMT	341,751@pthreads.cpp	8098	0.4877603215	0.0467830841	3.125	0.4470149254
freqmine	FM:FPArray_conditional_pattern_base	309,310@fp_tree.cpp	7391	0.5541804526	0.0376355603	0	0.1392405063
swaptions	SW:HJM_SimPath_Forward_Blocking	73,154@HJM_SimPath_Forward_Blocking.cpp	7364	0.2764791807	0.0005751513	9.0909090909	0
freqmine	FM:FP_tree::insert	949,966@fp_tree.cpp	6917	0.8277618173	0.0500976234	0	0.0273224044
freqmine	FM:FPArray_scan2_DB	350,381@fp_tree.cpp	5720	0.5048423569	0.0437303697	0	0.351758794
rtview	RT:Context::renderFrame	66,702@render.cxx	3174	0.9039223656	0.0011769807	19.943019943	0.0891719745
fluidanimate	FA:ComputeForcesMT	214,853@pthreads.cpp	2783	0.3657655262	0.0428191841	0	0.2899628253
freqmine	FM:transform_FPTree_into_FPArray	155,166@fp_tree.cpp	2534	0.4815067513	0.0358818723	0	0.4614886731
bodytrack	BT:ImageMeasurements::InsideError	46,109@ImageMeasurements.cpp	2062	0.3856999884	0.0002790583	18.1818181818	0.000602047
rtview	RT:std::map	154,985@stl_map.h	1862	3.3720343532	0.0230309423	0	0.0043050431
rtview	RT:std::map	154,985@stl_tree.h	1718	3.4081854401	0.0241397472	0	0.0044368601
swaptions	SW:HJM_SimPath_Forward_Blocking	73,162@HJM_SimPath_Forward_Blocking.cpp	1644	0.3500613144	0.000172117	0	0
freqmine	FM:FP_tree::FP_growth	1241,1525@fp_tree.cpp	1634	0.3866509282	0.0025129529	0	0.0217391304
bodytrack	BT:ImageMeasurements::EdgeError	35,64@ImageMeasurements.cpp	1572	0.4331032842	0.0041584321	18.1818181818	0
swaptions	SW:Discount_Factors_Blocking	392,395@HJM.cpp	1558	0.3012216265	0.0038717263	0	0
bodytrack	BT:ImageMeasurements::EdgeError	35,71@ImageMeasurements.cpp	1494	0.3710617909	0.0014191062	18.1818181818	0.0002366304
ferret	FT:findLoop	545,607@emd.c	1366	0.7222204974	0.059575519	0	0
rtview	RT:Context::renderFrame	78,703@render.cxx	1280	0.7431274816	8.57485851483E-005	14.4578313253	0.3823529412
facesim	FS:PhysBAM::DEFORMABLE_OBJECT	24,377@DEFORMABLE_OBJECT.cpp	1269	0.3710384528	0.0003071442	7.4626865672	0.1705913134
ferret	FT:findBasicVariables	342,364@emd.c	1258	0.9092810071	0.0682310636	0	0
rtview	RT:Context::renderFrame	183,616@render.cxx	1256	0.4226514098	0.0090270812	6.7264573991	0
ferret	FT:LSH_query_bootstrap	217,257@LSH_query.c	1246	0.8241507871	0.0536725933	0	0.304652645
ferret	FT:russel	690,699@emd.c	1109	0.7031576651	0.0241507004	0	0.0606060606
rtview	RT:RTTL::BinnedAllDimsSaveSpace::recursiveBuildFast	46,784@BinnedAllDimsSaveSpace.cxx	1013	0.497790848	0.0048529832	13.7931034483	0.2108843537

Table 20: Balanced regions: Pot-pourri

Application	FunctionName	REF_XCLK	CPI	BMR	LMR
libm-2.12.so	__ieee754_exp	54.40 (9331)	0.4130141556	0.0032722003	0.5609756098
SYSTEM CALL	copy_user_generic_string	48.98 (24)	37.875	0	0.9010989011
libm-2.12.so	__ieee754_log	39.61 (896)	0.6709184771	0.0326533273	0.0212765957
SYSTEM CALL	compaction_alloc	34.58 (37)	1.250734574	0.0008532423	0.3718244804
SYSTEM CALL	__brk_limit	25.58 (11)	0.8375796178	0	1.5
libc-2.12.so	__nrand48_r	25.00 (53)	0.2973467521	0.0037792895	0
libc-2.12.so	__GI___printf_fp	25.93 (7)	0.619047619	0.0241935484	1.3333333333
libc-2.12.so	_IO_vfscanf	24.16 (803)	0.3432752871	0.0032326651	0
libc-2.12.so	__GI_____strtof_l_internal	24.01 (798)	0.528601144	0.0081367274	0
SYSTEM CALL	__brk_limit	23.96 (86)	0.7121702915	0.0002826456	0.1081081081
libm-2.12.so	__ieee754_log	22.80 (9859)	0.7318299228	0.0693529854	0
libm-2.12.so	__ieee754_exp	22.34 (265)	0.3630233212	6.37795777792E-005	0
SYSTEM CALL	compaction_alloc	21.35 (19)	1.2474048443	0.003236246	0.364806867
SYSTEM CALL	copy_user_generic_string	20.22 (18)	21.48	0	0.1779661017
libc-2.12.so	__GI_memcpy	20.56 (22)	1.6175115207	0	0.2150537634
SYSTEM CALL	copy_user_generic_string	18.60 (8)	22.3529411765	0	0.4242424242
libc-2.12.so	__drand48_iterate	18.87 (40)	0.2629834254	0	0
SYSTEM CALL	__brk_limit	16.33 (8)	0.8333333333	0	2
libm-2.12.so	__ieee754_exp	16.49 (7130)	0.3753622543	0.0004671902	0
libm-2.12.so	__dubsin	16.61 (158)	0.7928035982	0.0005068424	1.5
SYSTEM CALL	clear_page_c_e	15.38 (2)	0	0	0
libc-2.12.so	_IO_vfprintf	14.81 (4)	1.4941860465	0.0132450331	1
SYSTEM CALL	clear_page_c_e	12.50 (6)	24.4285714286	0	2
libstdc++.so.6.0.13	std::istream::sentry::sentry	10.42 (5)	0.3398058252	0	0
libstdc++.so.6.0.13	std::basic_streambuf	10.42 (5)	0.4843049327	0	0
libm-2.12.so	__ieee754_logf	10.65 (1826)	0.4657000501	0.0056524332	0.7741935484

Table 21: System Call intensive regions

3.4.6 System calls

Apart from regions within user code of programs, we also keep track of hot-spot system call regions with greater than 10% execution time within an application. Table 21 lists the regions, The numbers within brackets in the REF_CLK column indicate the number of samples and thus also indicate the execution time of the system call with respect to absolute time. Analysis of these system calls reveals that the exponential function and logarithmic computation are very expensive and are called from the PARSEC benchmark blackscholes. In terms of CPI, the copy function is very expensive is a good region to simulate performance optimizations with respect to page accesses and misses. With respect to last level cache misses the copy page system call is very expensive as expected.

4 Conclusions and Future Steps

In this document we have reported on the methodology for selection of interesting regions within programs for both variety in terms of processor resource demands and accelerators. We first detail the applications chosen and the proposed approach in Section 2. In Section 3, we apply the approach and discuss the interesting regions selected and their application in the project.

Future work will involve porting of the tools used in the approach to ARM architecture and then comparing the selected regions with the current list.

Acronyms and Abbreviations

- BMR: Branch Misprediction Ratio
- BT: bodytrack program
- CA: canneal program
- CPI: Cycles per Instruction
- CQA: Code Quality Analyzer
- DOE: Department of Energy
- FA: fluid animate program
- FM: freqmine program
- FT: ferret program
- LL: lulesh proxy application
- LMR: Last latency cache Miss Ratio
- LULESH: Livermore Unstructured Lagrange Explicit Shock Hydrodynamics
- MAQAO: Modular Assembly Quality Analyzer and Optimizer
- RT: raytrace program
- SC: streamcluster program
- SW: swaptions program
- Vec. Ratio. FP: Vectorization Ratio Floating Point
- VI: vips program

References

[Ban94] Prithviraj Banerjee. *Parallel algorithms for VLSI computer-aided design.* Prentice-Hall, Inc., 1994.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[BL09] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. 2009.

[BS73] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[CAP$^+$15] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):6, 2015.

[CRON$^+$14] A. S. Charif-Rubial, E. Oseret, J. Noudohouenou, W. Jalby, and G. Lartigue. Cqa: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[DBT$^+$07] Lamia Djoudi, Denis Barthou, Olivier Tomaz, Andres Charif-Rubial, Jean-Thomas Acquaviva, and William Jalby. The design and architecture of maqao profile: an instrumentation maqao module. In *Sixth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-6) March 11, 2007 In conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization, San Jose, CA*, page 13. IEEE, 2007.

[dOCKA$^+$14] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. Fine-grained Benchmark Subsetting for System Selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 132:132–132:142, New York, NY, USA, 2014. ACM.

[ESC05] Lieven Eeckhout, John Sampson, and Brad 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.

[GPU] GPUOpen. http://gpuopen.com/compute-product/lulesh.

[HJM90] David Heath, Robert Jarrow, and Andrew Morton. Bond pricing and the term structure of interest rates: A discrete time approximation. *Journal of Financial and Quantitative analysis*, 25(04):419–440, 1990.

[HKG11] RD Hornung, JA Keasler, and MB Gokhale. Hydrodynamics challenge problem. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.

[Jef13] Brian Jeff. big.little technology moves towards fully heterogeneous global task scheduling. In *ARM White Paper*. ARM, 2013.

[Jol] Ian Jolliffe. *Principal component analysis*. Wiley Online Library.

[K+12] Ian Karlin et al. Lulesh programming model and performance ports overview. 2012.

[KKN13] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. 2013.

[Lab] Lawrence Livermore National Lab. https://codesign.llnl.gov/lulesh.php.

[LJW+06] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: a toolkit for content-based similarity search of feature-rich data. *ACM SIGOPS Operating Systems Review*, 40(4):317–330, 2006.

[LUL] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[Mac02] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[PHC03] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244–255. IEEE, 2003.

[QD02] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX conference on File and storage technologies*, pages 7–7. USENIX Association, 2002.

[R C13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[Wik] Wikipedia. https://en.wikipedia.org/wiki/k-means_clustering.