ELI-C: A Loop-level Workload Characterization Tool

Elie M. Shaccour
American University of Beirut
Beirut, Lebanon
ems10@aub.edu.lb

Mohammad M. Mansour
American University of Beirut
Beirut, Lebanon
mmansour@aub.edu.lb

***Abstract*—Every processor manufacturing cycle includes a workload gathering step. The input to this step is a diverse set of workloads from processor-specific domains. Simulating the entire workload set is time consuming and resource intensive, and therefore is infeasible. To reduce the simulation time, some techniques in the literature have exploited the internal program repetitiveness to extract representative segments. Other approaches involve exploiting the program similarities and reducing the number of workloads. In this paper, we propose an orthogonal and complementary approach to reduce the simulation time by exploiting loop-level program characterization similarities across workloads in a set. We developed a loop-level analyzer, dubbed ELI-C, that extracts loop characterizations from workloads, and then applies a machine learning technique to group workloads based on similarities. Finally, we employ a novel heuristic to exclude workloads that are highly similar and arrive at a smaller representative subset that preserves the characteristics of the initial set. Simulation results demonstrate that applying this technique to a set of 20 workloads reduces the set size into half while preserving the main characterization of the initial set of workloads.**

Keywords*—*Loop-level analysis, performance, workload characterization.

I. Introduction

A processor is typically optimized for its targeted application domain. The design phase begins by choosing the reference workloads for which the processor is targeted for. For example, if a processor is targeted for media usage then the set of reference workloads would include media applications like video decoders, signal-processing and other relevant workloads. Since most domains span a wide range of applications, we end up with a large workload set. Furthermore, a processor currently at the beginning of manufacturing will not be released for at least three years. So the previously chosen workloads might become obsolete when the processor is released. To overcome this situation, computer architects try to predict future usages within the targeted domain and add them to the workload set. In brief, the workload collection stage generates a large workload set to understand and optimize the processor design. To narrow down the design features, computer architects analyze and characterize each benchmark's execution behavior. For example, identifying high data sharing numbers might lead to designing a larger common L_2 cache for the various cores rather than higher independent L_1 caches. Once the general processor features are decided (memory hierarchy, number of cores, etc.), a simulator will be developed to pin down the final properties, *e.g.* L_1 cache size 1MB instead of 2MB. For this, the workload set needs to be simulated with each configuration to determine the best possible outcome.

To discover optimization opportunities, the developed simulator collects detailed simulation results from the chosen workloads and as such requires a lot of processing power and time. Some simulations can take up to days and even months to completion thus making it unfeasible to run all the workload set with each configuration. To reduce simulation time, architects can resort to either a less detailed simulation or a reduced workload set. While lower detailed simulations lead to less fine-grained results, a reckless reduction in the simulation set might lead to completely hiding-out certain results. Recently, researchers targeted the latter solution with systematic reduction techniques to identify the similarity. Yet, what is similarity? And how is it defined between workloads? Even though each of the applications might differ by description and by the tasks they perform, it might be evident that they interact in a similar approach with the processor such that analyzing and optimizing for only one of them would be sufficient to drastically improve the execution of the other. Being able to identify this similarity and thus remove redundant workloads would reduce the workload set and thus simulation time. For example, our previous media set might include both an *MPEG* codec for videos and a *JPEG* codec for images. Although they have different described applications, yet MPEG includes a JPEG codec, which might seem redundant in this set.

In this paper we propose a loop-based characterization tool, *ELI-C*, to systematically analyze and reduce a workload set. The proposed framework starts by identifying and characterizing key loops from the workloads in the set. Using the characterizations, we construct a feature vector for each loop and feed these vectors into a machine learning tool which classifies them into clusters based on similarity. Remapping the workloads into a vector of these cluster centers provides a standardized way to represent these workloads so that they can be compared. We define a similarity score between two workloads as the inner product of the two workload vectors. Finally, we start to eliminate similar workloads one by one using a heuristic that works with the similarity scores. By using this framework we were able to achieve a 50% reduction in the number of applications while preserving dynamic characterization techniques.

II. Background and related work

Workload characterization has been employed to understand the execution behavior of multiprocessor workload projects. Due to the power wall limitation, the trend of

This research has been supported by Intel Corporation under the Middle East Energy Research collaboration project.

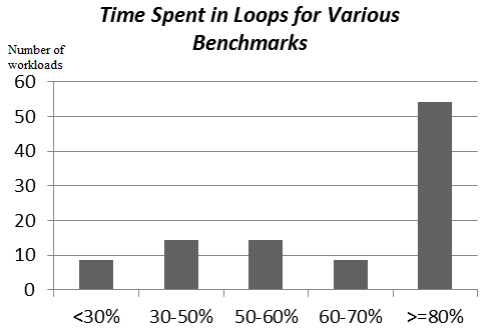


Fig. 1: Loop time of various scientific workloads

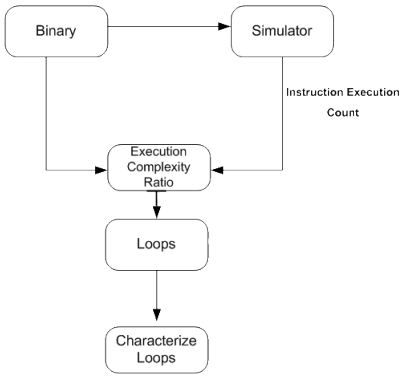


Fig. 2: ELI-C data flow graph

microprocessor development has shifted from a faster single processor core to the integration of multiple cores on the same die. However, building many core simulators is a challenging process and very few architectural simulators are designed to take advantage of the multiple cores offered in today's processors: for example MIT Graphite simulator [1]. To manage simulation time and resource challenges for multiprocessor development projects, two complementary approaches are often taken to reduce the target workloads.

The first approach is to reduce the simulation time of each application by only running smaller yet representative portions of the application. Sherwood et al. [2] presented a methodology to extract a reasonable sized interval from the program which has a similar fingerprint to the full program. This is achieved by building the basic block vectors (BBV) of the program. The BBV contains the normalized execution frequency of Basic Blocks (single entry, single exit parts of the code). After isolating the initialization section, clustering is performed on the remaining part of the BBVs to find the reduced set. This method exploits the internal similarity/redundancy of the programs, and aims at reducing the simulation time by using the representative program phase as the proxy of the entire program execution. The SMARTS framework proposed by Wunderlich et al. [3] employs a statistical sampling strategy to predict a given cumulative property (*CPI*) of the entire workload with a desired confidence level. It uniformly samples the program intervals in the dynamic instruction stream for detailed simulation, and uses fast-forwarding and warm-up on the discarded instructions to speed up the simulation time.

The second approach, which relies more on characterizations, is to identify the similarity between the workloads and remove redundant programs to yield a smaller representative subset. Biena et al. [4] presented a comprehensive comparison between PARSEC and SPLASH-2 benchmarks. Their study used execution driven simulation on PIN to obtain chosen characteristics including instruction mix, working set sizes, and sharing size and intensity. Using the collected data, they applied Principle Component Analysis (PCA) to choose the uncorrelated data and then applied hierarchical clustering to group similar programs into single clusters. Phansalkar et al. [5] explored the benchmark similarity inside SPECcpu 2006 benchmark suite to identify the distinct and representative programs. They used micro-architecture dependent hardware counters to collect program characteristics. Then, in a similar methodology to Biena et al. [4] they used PCA to extract uncorrelated dimensions from a diverse set of workload characteristics, and finally they measured the program similarity with K-means clustering analysis. It is good to point out that in [4] they used PIN, a detailed software simulator, while [5] exploited the limited view hardware counters approach.

Joining both previous approaches, Lieven et al. [6], proposed a methodology that exploits both the program's internal repetitive phase behavior and cross-program similarities. They used SIMPOINT [7], a program phase analysis tool, to break each program into intervals and collect their microarchitecture-independent characteristics. Representative phases from the entire benchmark suite are collected. This methodology reduced the number of simulated instructions by 1.5 over SIMPOINT. Our proposed framework, similar to Lieven [6] tries to exploit internal as well as cross-program repetitiveness. However, rather than being solely based on execution dynamics, our framework integrates analysis of critical program structures such as loops which not only provides additional views, accuracy and lower simulation time, but also opens up a new possibility to explore emerging workloads using the combination of these structures. A number of researches have looked at loop-centric analysis. Moseley et al. [8] presented a set of loop-centric profiling tools: an instrumentation-based loop profiler, which uses basic blocks to detect and account the loops; and a light-weight sampling based loop profiler, which has a faster profiling speed but provides less detailed information. The goal of their profiler was to help exploit loop level parallelism and expose the program hotspots. Our loop level analysis differs from other loop-centric analysis by having the capability of extracting characteristics such as arithmetic intensity, instruction mix, branch misprediction ratio, data reuse patterns, etc.

III. Characterization Framework

Traditional characterization approaches either execute every instruction of the workload or sample periods of execution. These approaches disregard workload behavior characteristics. In this work, we studied execution trends of several open source workloads: *NPB*, *PARSEC*, *HPCC*, etc. This study led us to perceive that scientific workloads spend the majority of their time in loops, Fig. 1. Loops are repetitive and execute one of many instruction paths over the various iterations. To reduce

simulation time, we identify and characterize a single iteration of the most executing paths of a loop body (different control flows constitute different loops) hoping it will be enough to understand the execution behavior of the entire workload.
The execution path of our characterization tool *ELI-C*, *efficient loop identifier and characterizer*, is shown in Fig. 2.
We restrict our tool to binary analysis as it presents the most noise reduced form of what will execute on the processor. In the simulator phase, the binary will be executed simultaneously on both Gprof [9] and Valgrind [10]. Gprof is used to calculate the absolute execution time spent in each function, while Valgrind's internal tools Callgrind and Cachegrind are used to extract instruction execution count as well as cache misses for various cache sizes, respectively. Using both the binary and the instruction execution count, loops are identified with a novel developed ratio, *execution complexity ratio*:

$$ECR = \frac{AEC}{TEC} : \begin{cases} <1, & \text{conditional statement} \\ =1, & \text{serial code} \\ >1, & \text{loop code} \end{cases} \quad (1)$$

where *AEC* is *actual execution count* or number of times the assembly line executed while the *TEC* is the *theoretical execution count* or number of times this assembly line would have executed if it was part of a serial code , *i.e.,* number of times the function has been called. Once loops are isolated, the following characterizations will be collected.

A. Direct Characterizations

Direct characterizations are those computed cumulatively across each loop instruction without depending on other instructions.

- **Loop iterations** Number of times the loop body was executed.
- **Static instructions** Number of assembly instructions executed in the loop in a single iteration.
- **Relative size** Static instructions aren't indicative of the weight of the loop since a single-instruction loop executing a million times could be performing more computations than a 1000-instruction 10-iteration loop. This characterization shows the relative size of the dynamic number of instructions executed by the loop relative to the total number of instructions executed by the workload , *i.e.,* $\frac{(LoopIterations)*(StaticInstructions)}{(DynamicNumberOfInstructionsExecuted)}$.
- **Scalar instructions** Number of single-operation, single-data instructions.
- **Vector instructions** Number of single-operation multiple-data, *SIMD*, instructions. This identifies parallelism in the code.
- **Floating Point instructions** High latency instructions, some of which take hundreds of cycles. It is essential to keep track of these instruction to help us understand the bottleneck of the loop.
- **Bytes (raw)** Previous characterizations dealt with the computational aspect of the loop. Memory is the other crucial aspect to look at in the characterizations. The virtual memory system starts with cache and goes up to the disk storage, this characterization presents a sum of all the data requests made by the loop to the virtual memory system. This characterization is further divided between bytes-loaded and bytes-stored.
- **Memory accesses** Raw bytes represents the data requests in bytes. Absolute numbers *e.g.* 128 bytes, might not be useful since a single instruction might load the 128 bytes or it might be a one-byte load instruction executed 128 times. To better represent this number, we split the number of instructions responsible for the data loads and stores. This characterization is further divided into scalar loads, scalar stores, vector loads and vector stores.
- **Bytes (filtered)** Another problem with the raw byte characterization is that it doesn't represent the behavior of the loop in today's architecture with caches. Filtering the references through the cache allows us to learn about data reuse in the loop. The tool is equipped with a configuration file to allow the user to simulate for the needed cache configurations.
- **Branch mis-prediction** It has been previously mentioned that the number of loop iterations is an *average* representation. This is due to the fact that a conditional loop might not always be executed or may be executed a different number of times. Given that each loop execution might have a different control flow graph, this characterization helps us know the confidence with which this loop was executed.

B. Indirect Characterizations

Indirect characterizations require an extended view of previous and latter instructions to help classification.

- **Instruction Classes** Classifying the instruction by type requires tracking the registers and their ultimate use. The tool creates a buffer for each loop and keeps track of registers and the way they are used. A careful breakdown of each instruction had to be taken into consideration as some instructions don't have a destination register (*j*), while others have the first operand also a destination (*add*), and the rest have a separate register for destination (*mov*). The following is a description of the classes of instructions:
  1) **Address calculation instructions** Architectural register limitations presented by the x86/64 architecture is compensated by frequent trips to the memory and by providing more physical registers. The x86/64 provides complex memory addressing modes and thus some instructions, considered as overhead instructions, are used to calculate the memory addresses [11].
  2) **Control instructions** Number of execution path changing instructions that are part of the loop, *e.g.* jumps and calls. This characterization helps reveal, among other details, if this loop is part of another nested loop.
  3) **Compute instructions** After removing the previous overhead instructions, we are left

with pure compute instructions that are used to perform the high level operations they are assigned for. This characterization helps us know the size of the computations performed in the loop.

- **Memory Access Patterns** The main factor that effects the cache hit rate, *bytes (filtered)*, is the memory access patterns. For example, if we have a loop that accesses a fixed word-length stride pattern then cache misses decrease since a miss is followed by a cache hits if cache-line size is greater than one word. To discover memory access patterns, the tool keeps track of the static addresses of both the reads and the writes, separately *e.g.* address *[esp - 5]* is stored as is rather than replacing *esp* by its address. Although this hides out some indirect patterns yet it reduces the analysis time. Our pattern detection algorithm is then run over the archived load and store addresses. It will detect any pattern that is repeated more than 2 times and could be of any frequency, *e.g.*, -5,-5,-5, or -5, +6, -5, +6, etc.

IV. Elimination Scheme

Starting with a set of workloads, our target is to get a reduced set while maintaining the characterizations of the initial set. In brief, we achieve this procedure by:

1) Simulating workloads on *ELI-C*
2) Creating a representative vector for each loop
3) Joining loop characterizations from the same workload to get a workload-specific signature vector
4) Calculating pair-wise workload similarity
5) Applying an elimination procedure on the workload set

Each of the previous steps is refined to remove biasing and improve the resulting outcome as discussed below.

A. Simulation

The first and simplest step is simulating the workloads using *ELI-C* to get loop-level characterizations.

B. Data Preprocessing

As in any general data mining problem, the simulated data set is preprocessed to remove noise and outliers. Preprocessing is carried out in two complementary ways. In a general view, the data we begin with can be considered as a *k*n* matrix, we will reduce the number of rows by removing outliers, through *data cleaning*, and the number of columns by reducing the data dimensions, *data reduction*.

1) Data Cleaning: Data cleaning is divided into 2 phases. The first phase occurs before integrating data from the various workloads into a single data structure. For each simulated workload we calculate the loop execution time relative to the entire workload's execution. Since our target is to have the loop characterizations representative of the workload, then workloads with low execution time are removed. In the second part, we look at loops from each workload separately. Not every loop has significant execution time. Significant loops, in our case, are those with relative execution time greater than 0.1%. Anything below this number is considered redundant and thus removed.

2) Data Reduction: Collected characterizations are split between architecture dependent and independent. Correlation between characterizations results in data biasing and ultimately a biased classification. Correlation is difficult to detect and remove manually. Principle component analysis, *PCA*, is a common applied algorithm that removes correlation and reduces a data set's dimensionality. Another source of data biasing is having characterizations on a different scale. To overcome this, we normalize all the characterizations. The output of normalization is a matrix, whose numeric values range between *0* and *1*. The input to PCA is the normalized matrix in which the rows are the set of reduced loops and the columns are the collected characteristics. PCA computes new variables, called principle components *(PC)*, which are a linear combination of the initial characteristics. The new characteristics, PCs, are uncorrelated, independent and used to represent the loops in the classification algorithm. However, the new PCs also have different ranges, so we normalize the characterizations again.

C. Data Classification

The normalized PC matrix contains representative features of each characterized loop. Our target is to classify similar loops together. We use an *Expectation Maximization* (EM) algorithm on the popular WEKA data mining toolkit [12]. The popular *k*-means algorithm is a specialized form of EM. EM was chosen since unlike *k*-means the algorithm will decide on the best number of clusters. EM starts with an initial set of parameters and iterates until the clustering cannot be improved, that is, until the clustering converges or the change is sufficiently small. The *expectation* step assigns objects to clusters according to the current fuzzy clustering or parameters of probabilistic clusters. The *maximization* step finds the new clustering or parameters that maximize the sum of squared error (SSE) in fuzzy clustering or the expected likelihood in probabilistic model-based clustering. Once completed, each loop will have an additional characterization specifying its cluster membership.

D. Elimination

At the moment, we have a set of clusters containing similar loops together. However, since our target is to remove similar workloads and not loops then we'll need to form a higher level of abstraction from the current loop-level view that we have. So before starting elimination, we iterate over the characterized loops to form a workload-specific feature vector. This feature vector *(FV)* will represent the time spent by the workload in each of the clusters as follows:

$$FV_{Workload_i at Cluster_n} = Workload_i \cap Cluster_n \quad (2)$$

Since each workload has a different execution time value, we normalize its clusters' execution time by its total execution time. Once completed we obtain the blueprint of each workload using which we are able to compare workloads. The following subsections introduce the elimination scheme step-by-step.

1) Similarity Score: To recognize similar workloads we defined a similarity metric between workloads as the inner product of the two benchmarks' representative vectors. The similarity score has a range between *0* to *1* and is mathematically calculated as follows:

$$Similarity(BM_m, BM_n) = \sum_{i=0}^{Clustersize} BM_{m_i} \cap BM_{n_i} \quad (3)$$

2) Similarity Matrix: To visualize similarity between workloads we construct a 2D NxN symmetrical matrix containing all the workloads such that elements (p, q) and (q, p) in the matrix have the same value. The similarity matrix will be used to pick the tuple of workloads that has the largest similarity value, one of which will be eliminated.

3) Elimination Strategy: To eliminate similar workloads we iterate through the similarity matrix, each time picking the workloads with the highest similarity *, i.e.,* the largest element in the matrix. Since both workloads are similar then they are representative of each other and we could eliminate one of them without effecting the dynamic execution characteristics of the workload set. To decide which of the two workloads we eliminate, we calculate each workload's similarity with the rest of the workloads in the set *, i.e.,* for the workload at (*p*,*q*) its similarity with the remaining set is the sum of all the values in row *p* or column *q* excluding its own value. The workload representing the lower value of both resultants is then eliminated and so we maintain higher chances of further redundancy and elimination in the set.

4) Elimination Limit: Loops in the same cluster are similar and fall within short distance of each other. To maintain the workload set's dynamic characterizations, we'll need to maintain a certain number of loops within each cluster. Our stopping criteria monitors the number of loops within each cluster across elimination iterations. Once an elimination round causes any of the cluster sizes to drop below 50%, the removed workload is re-inserted and elimination stops. The threshold value of 50% was chosen to retain a balance between the remaining workloads and the set's characteristics.

5) Validation: To verify that the chosen subset of workloads is representative of the initial set and maintains its behavior, we compare 3 dynamic execution aspects between the initial and final set on different processor technologies. The used validation metrics are: *cycles-per-instruction* (CPI)*, scalability, data sharing* and were collected using Intel Sampling Enabling Product (SEP) tool version 3.5 from Intel VTune Amplifier XE 2011 [13].

V. Simulation

A. Workload Set

The initial workload set consists of workloads with high loop execution time, Table I. The selected workloads were compiled using Intel's C++ Compiler from the Composer XE suite 2011 [14]. The workloads were then executed and analyzed on a system with Intel Xeon E7-4860 processors. Workload specific signatures vectors were processed by the WEKA tool [12] using *EM* clustering algorithm.

TABLE I: Input workload set

Workload	Loop Execution Time	Number of Loops
NAS CG	98%	24
NAS EP	99%	5
NAS FT	95%	15
NAS LU	98%	12
NAS MG	94%	21
NAS SP	93%	40
NAS UA	87%	106
PARSEC Freqmine	89%	37
PARSEC Streamcluster	99%	15
Lammps	89%	16
MRI Reconstruction	90%	24
Finite Element Solver	94%	116
Mini MD	96%	12
Splash Water N-squared	98%	149
NU Bench Kmeans	89%	13
HPCC Gups	96%	6
HPCC Singlestream	97%	7
HPCC StarRA	99%	5

B. Experimental setup

To insure fair workload-to-workload comparison we optimized all the workloads with similar compilation flags for speed, vectorization and enabled aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. To validate that the new workload subset is representative of the initial set we used three different generations of Intel processors:

1) **Intel Core i7-2600K** A 64 bit 4 cores, 8 thread processor running at 3.40 GHz. Equipped with Turbo boost 2.0 technology that can boost clock speed up to 3.80 GHz. It includes an 8MB Smart cache system and a 5GT/s system bus.
2) **Intel Core i7-975** A 64 bit 4 cores, 8 thread processor running at 3.33 GHz. Equipped with a Turbo boost technology that can boost clock speed up to 3.6 GHz. It includes an 8MB Smart cache system and a 6.4 GT/s system bus.
3) **Intel Core 2 Extreme CPU X9650** A 64 bit 4 core, 4 thread processor running at 3.00GHz equipped with a 12MB L2 cache and a 1333MHz system bus.

C. Data Preprocesing

Given the workload set, we gathered the loop characterizations in a single file for preprocessing. This helped us eliminate outlier loops, those having low execution time, and thus decreasing the set's size from around 3000 to 623 loops constituting the final loop set. The data set is then transformed by normalizing all the characterizations. Now that all characteristics are equally weighted, we apply PCA reducing data dimension from 13 characteristics to 11 uncorrelated characterizations. The set was again normalized, leaving us with a set of 623 loops with 11 equally weighted characterizations in addition to loop-time and the workload it belongs to. This data is now ready to be clustered.

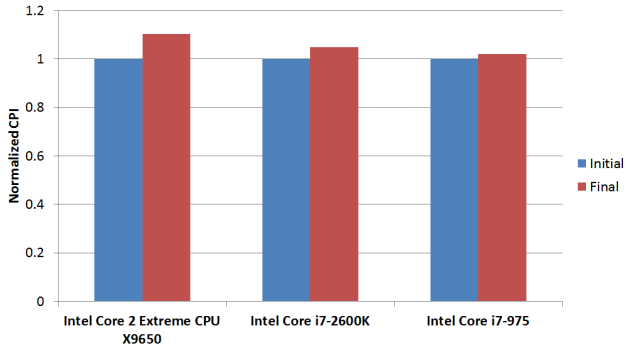
D. Data Clustering

Using EM algorithm on WEKA [12], the loops are clustered and distributed among 6 clusters. The output distribution wasn't uniform as cluster 4 joined loops from most workloads,

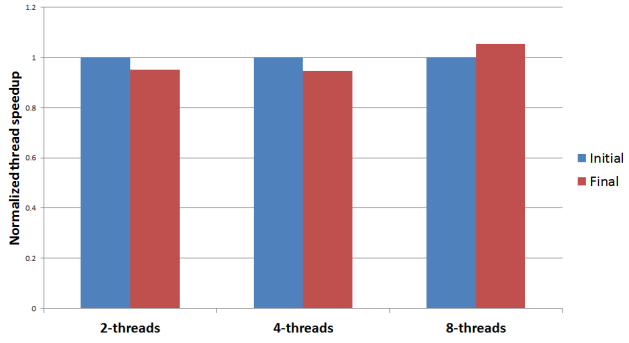
while both clusters 1 and 2 were highly concentrated in single workloads.

E. Elimination

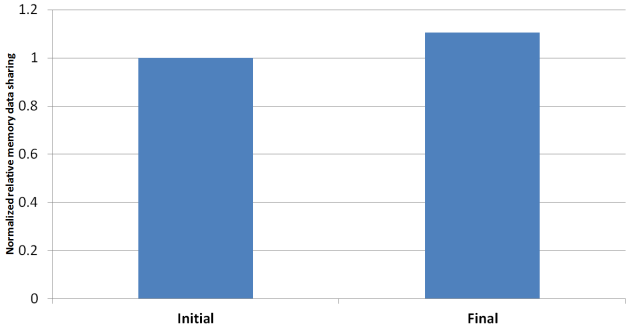
Elimination started over the similarity matrix, at each iteration choosing the workload tuple with highest similarity, then calculating their similarity with the rest of the workload set and removing the one with lower value. After removing the 10^{th} workload, one of the clusters' size dropped below the 50%, our predetermined cluster size threshold. Thus we kept the 10^{th} workload and ended up with a final set of size 9.



(a) CPI comparison



(b) Thread speedup comparison



(c) Data sharing comparison

Fig. 3: Dynamic characteristic evaluation of reduced set compared to initial set

F. Validation

The elimination scheme was able to reduce the set by 50% of its size. To verify that the reduced set is representative of

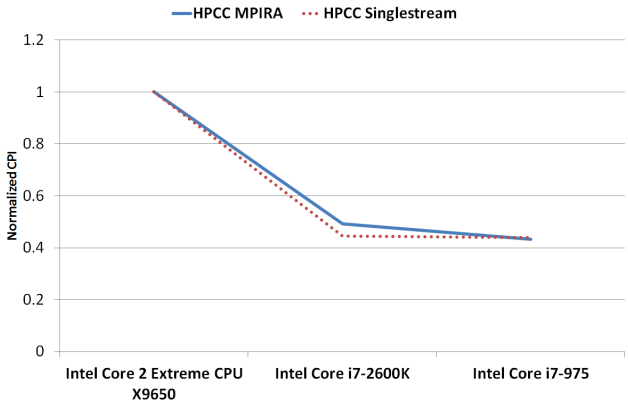


Fig. 4: HPCC bottleneck on older Intel generations

the initial set, we executed the workloads on the previously mentioned Intel processors and collected CPI, sharing data, and total execution cycles when executed with 1, 2, 4, and 8 threads. The characteristics of the final set was then compared to the initial set, as shown in Fig. 3. Numerically, the CPI varied between 10%, 4%, 1% on Intel Core 2 Extreme CPU X9650, Intel Core i7-2600K, and Intel Core i7-975, respectively as shown in Fig. 3(a). The highest difference was spotted on the Intel Core 2 Extreme CPU X9650. After studying workload behavior on the processors, we found that Intel Core 2 Extreme CPU X9650 is the oldest technology between those tested and some of the workloads showed slow execution. This became more clear by comparing the CPI of HPCC's MPIRA and SingleStream on the 3 generations. As shown in Fig. 4, the CPI was reduced more than half between both generations. Such results are expected since the collected data is from simulations on newer processors. The speed-up between *1 and 2*, *2 and 4*, *4 and 8* threads on Intel Core i7-2600K varied between 4% and 5%, as shown in Fig. 3(b). Finally, the comparison of the data sharing, Fig. 3(c), was around 10%. This difference could be contributed to the tool's lack of any collected data sharing characteristics.

VI. Conclusion

In this paper a novel systematic framework to subset workloads based on program structure characteristics and machine learning techniques has been presented. Based on the observation that loops are the key building block for most workloads, a tool to extract loop specific characteristic has been proposed. The characteristics were used to from a feature vector that provides a signature for each loop. Processing all such vectors through machine learning algorithms, unique loop structures that are the building blocks of all loops were identified. Remapping workloads into the unique set of loop structures generated a signature for the workloads. Using a systematic elimination process on the similarity scores, the initial set of workloads was reduced to a smaller representative subset. To validate our framework, the framework was applied on a set of 18 workloads from different domains and showed that the reduced set of workloads preserves the execution characteristics of the initial set over three generations of processors. This approach provides more accurate subsets than previous sampling techniques, solely based on the fact that loops are the

essential driving engine for most compute intensive workloads, thus they should be driving the subsetting algorithm. In the future, more characteristics could be added to span wider characteristics and take sharing data into consideration.

REFERENCES

[1] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[2] 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*. IEEE, 2001, pp. 3–14.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 84–95.

[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*. IEEE, 2008, pp. 47–56.

[5] 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*, vol. 35, no. 2. ACM, 2007, pp. 412–423.

[6] 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*. IEEE, 2005, pp. 2–12.

[7] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. IEEE, 2003, pp. 244–255.

[8] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 143–152.

[9] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[10] J. Seward, N. Nethercote, and J. Fitzhardinge, "Valgrind, an open-source memory debugger for x86-gnu/linux," *URL: http://www. ukuug. org/events/linux2002/papers/html/valgrind*, 2004.

[11] I. Intel, "Intel 64 and ia-32 architectures software developers manual," *Volume 1, 2A, 2B, 2C, 3A, 3B and 3C*, 2013.

[12] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[13] I. V. A. X. 2011.

[14] "Intel c++ compiler, composer xe suite 2011," http://software.intel.com/en-us/articles/intel-compilers/.