Analyzing the Impact of On-chip Network Traffic on Program Phases for CMPs

Yu Zhang, Berkin Ozisikyilmaz, Gokhan Memik, John Kim and Alok Choudhary
EECS Department
Northwestern University
Evanston, IL, USA
{yzh702, boz283, memik, jjk12, choudhar}@eecs.northwestern.edu

Abstract—**It is known that the execution of programs exhibits repetitive phases; in other words, the execution of programs can be partitioned into segments of execution, during which the application exhibits unique architectural properties. This property has been used for various optimization goals. In addition, phase information is utilized to reduce the run time of the architectural simulation. Conventionally, an application is examined in an architecture-independent manner (such as the number of times a basic block is executed) to extract information about the phases and then only the representative execution intervals are executed to analyze architectural choices. We claim that such approaches are becoming inadequate in the many-core era as application execution is not dominated by the instructions only, but instead the communication structure of the application is becoming as important as the instruction behavior. Hence, we propose to utilize communication behavior to determine the phases of an application. Our results reveal that the inclusion of the communication information can increase the accuracy of the phase detection significantly. Specifically, for SPLASH2 and MineBench applications, the average (geometric mean) CPI error rate with the instruction-based phase detection is 11.01%, while our phase detection scheme has an average error rate of 3.41% when compared to the simulations that run the applications to completion.**

I. INTRODUCTION

Previous works have shown that programs exhibit repetitive behavior at different execution intervals. Based on this observation, the terminology *phase* is defined. A phase is a set of intervals within a program's execution that exhibit similar behavior. The intervals are not necessarily temporally adjacent. In other words, a phase can reoccur at different parts of the execution. The information about phases has been used for various purposes. Dhodapkar and Smith [1-3] use it for reconfiguration of the multi-configuration units to save energy. Balasubramonian et al. [4] use phase information to guide dynamic cache reconfiguration to save energy without sacrificing performance. Merten et al. [5] use the phase information to develop a run-time system for dynamically optimizing frequently executed code. Barnes et al. [6] extend this idea to perform phase-directed compiler optimizations. Biesbrouck et al. [7] use phase behavior to guide simulation for Simultaneous Multithreading [8]. One of the most common means of utilizing the phase information is to reduce the architectural simulation time. Specifically, if representative intervals can be extracted from the execution, an architecture can be simulated for those intervals only (rather than running the application to completion, which takes considerable simulation time) without causing any inaccuracy in the simulation results. Such approaches reduce the overall simulation time by orders of magnitude. For example, with an interval of 10 million instructions, no more than 30 simulation points are needed to represent the complete execution of SPEC 2000 benchmarks, for which the whole execution may take over 100 billion instructions [9, 10]. Simulating only these carefully chosen simulation points can save hours to days of simulation time with very low error rates.

The phase analysis for serial programs is already a mature area [1, 6, 10-17]. Architectural metrics (for example, CPI, cache miss rate or hit rate, branch frequency) and code signatures have been examined to extract phase information. Results show that the extraction of phases through such means is accurate. However, as the manufacturers are moving towards chip-multiprocessor (CMP) designs, the accuracy of such approaches may degrade. As technology scaling continues to increase the amount of transistors available, recent trend has been to increase the number of cores on a chip instead of adding complexity to a single core. Many manufactures, including Intel, IBM, Sun, and AMD, have produced microprocessors that incorporate multiple processing cores. The increasing number of cores has resulted in the on-chip network connecting the cores together becoming a crucial component in determining the overall performance of the processor.

As a result, techniques that determine phases without the consideration of the communication behavior of the application can generate relatively high error rate. In Section VI, we present experimental results showing that a typical phase detection approach that considers only instruction behavior causes over 11% inaccuracy on average. To remedy this limitation, we propose to utilize on-chip communication information to determine the execution phases. We introduce a vector related to the on-chip interconnect network traffic as an additional representation of the program execution. This vector is a one-dimensional array, with each array element recording the traffic information of the on-chip router. Then, we combine this structure with the instruction count based structure to represent the execution of program on CMP machines. Program phases are then detected based on these new arrays. Experimental results show that incorporating the communication behavior reduces the CPI error rate from 11% to 3% on average. As the number of cores on a chip scales, it is expected that the interconnection network will become an even more important determinant of the overall performance; hence the motivation to utilize our approach will increase.

This paper is organized as follows. In Section II, we overview the related work. In Section III, we present the details of our methodology. Section IV describes the technique we use to profile the execution of the programs. Section V presents the experimental setup. In Section VI, we provide our experimental results. Section VII concludes the paper with a summary of our contributions.

II. Related Work

Several researchers have examined phase behavior in programs. In this section, we provide a brief description of studies related to phase identification and phase-based optimization.

Sherwood et al. [18] presented that programs have repeatable phase-based behavior over many architecture metrics, such as cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy for SPEC 95 Benchmarks. Repeating patterns were found in many of the programs, and the essential architecture metrics change similarly over time. They also proposed that phase behavior in programs could automatically be identified by only examining code execution [19]. They extended their work in [16] to develop automatic techniques that are capable of finding and exploiting the large scale behavior of programs. SimPoint [10] was developed to detect phase by using the clustering algorithm (K-means). In our work, we extend the use of SimPoint. Instead using the traditional Basic Block Vectors, we use new kinds of vectors to classify the intervals into clusters. We describe the SimPoint in more detail in the next section. Dhodapkar and Smith [2, 3] utilized the relationship between phases and instruction working set and observed that phase changes at the same time as the working set changes. This directed them to reconfigure the processor units to save energy.

Isci et al. [14, 20, 21] used hardware performance counters to exploit phase behavior in programs and proposed that the power phase behavior could be identified dynamically by using power vectors. Deusterwald et al. [22] utilized hardware counters and other phase prediction schemes to detect phase behavior. Liu et al. [23] dynamically detect phase behavior by tracking procedure calls using a call stack. Davies et al. [13] used Intel's VTune Performance Analyzer to get a representation of program execution. Sampling information is extracted from VTune at runtime on native hardware. Then, simulation points are generated similar to SimPoint. Annavaram et al. [11] employed this VTune approach to identify phase behavior for database applications. Perelman et al. [24] used the VTune sampling approach to collect Extended Instruction Pointer Vectors and then produced Sampled Basic Block Vectors to detect phase behavior of parallel applications on a shared memory machine.

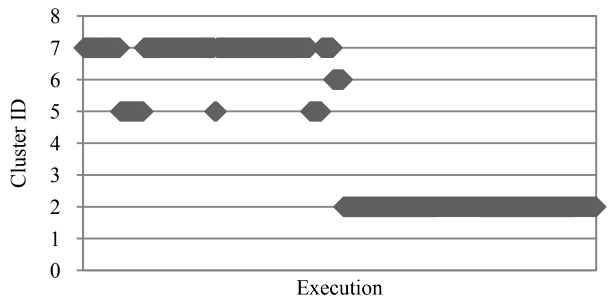
Analyzing phase behavior over different time scales has also been studied. Hind et al. [25] presented the impact of granularity and similarity on phase analysis and provided a framework to perform phase analysis appropriately. Lau et al. [15] examined the importance of varying interval length in phase identification. Vandeputte and Eeckhout [17] provided phase complexity surfaces to characterize a program's phase behavior across various time scales. Cho and Li [12] proposed an approach to quantitatively analyze the changing of phase dynamics across different time scales and presented a framework classifying phases which exhibit homogeneity in their scaling behavior.

Most of the above works are done based on one important assumption that the programs of interest are executed serially, except the work of Perelman et al. [24]. However, their phase behavior detection is done on shared memory architectures. In this paper, we propose an approach to examine the phase behavior in parallel programs on CMP machines.

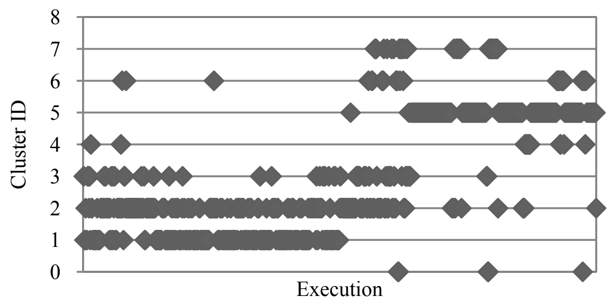
III. Methodology and Metrics

Phases can be extracted using information about basic blocks or instruction counts. A basic block is a single-entry, single-exit section of the code, which does not contain internal control flow. SimPoint [10] utilizes the Basic Block Vector (BBV) to create phases. BBV is a one-dimensional array. Each element in BBV represents the number of times a basic block is executed during an execution interval. BBV is a commonly used structure to represent the execution of a given interval [16, 24] and is an attractive representation of the execution. However, when the number of cores scales up, generating the basic block vector becomes complicated. There would be tens of thousands of basic blocks in the program, which means the number of elements in one array (one basic block vector) can exceed hundred thousand. Due to the implementation difficulty, we don't use BBV to detect program phases. Instead of BBV, we introduce the Instruction Count Vector (ICV), which can also be used on SimPoint. ICV is much simpler and more straightforward than the projection of BBVs. ICV is also a one-dimensional array, where each element in the array corresponds to the instruction count of a core (number of instructions executed by a core) within the execution interval. In this paper, the interval we use is based on global instruction count (total number of instructions executed by all of the cores). In our experiment, an interval contains one million instructions overall. Then, the number of instructions executed by each core is collected to form the ICV.

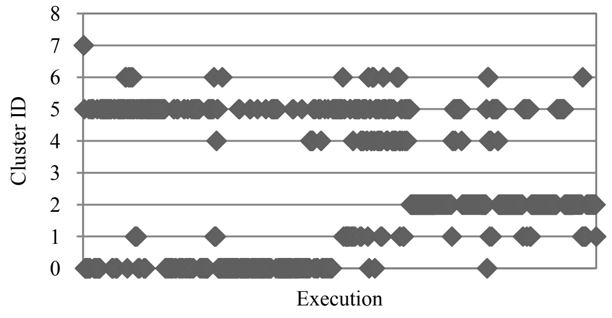
Note that the interval can neither be too small nor too large. A small interval means that to represent the whole execution, more simulation points are needed, or the error rate will increase. On the other hand, there is also requirements on the number of intervals. Hamerly et al. [10] describes that the number of intervals should be sufficient in order to make the clustering algorithm work properly. This number should be no less than a thousand to make sure that the clustering algorithm can provide a decent partition of the



(a) Phase detection based on instruction count



(b) Phase detection based on traffic information



(c) Phase detection based on combination of instruction and traffic information

Figure 1. Phase clustering of Barnes application based on different profiling information: (a) instruction count based clustering, (b) traffic based clustering, and (c) clustering based on combination of instruction and traffic

intervals. As a result, to guarantee adequate number of intervals, each interval should not be too large. For the target applications in our work, we found that the interval of one million instructions provides a good trade-off between these forces.

In addition to the ICV, in this work, we introduce Traffic Count Vector (TCV), which is also a one-dimensional array with each element corresponding to the traffic count (number of packets) going through each router. To make the traffic profiling more specific, we record the number of packets generated by each router, number of packets destined for each router and also the intermediate packets. Finally, we combine the ICV and TCV into one vector, where each element contains the corresponding values from the ICV and the TCV. We then detect phases using this combination of profiling information.

Figure 1 shows phase detections through three different methods: ICV-based phase detection, TCV-based phase detection and phase detection based on the combination vector. The results are presented for the Barnes application [26] (please see Section V for details of the application).

In Figure 1, the X-axis represents the execution of the program and Y-axis is the cluster ID. Each execution interval is assigned a unique cluster ID by the K-means clustering algorithm. Figure 1 implies that phases detected by TCVs are similar to the ones extracted by ICVs in the macro scope but still they are different in detail. Based on the calculation, the simulation points selected by the combination vector can cover more than those selected by ICVs.

We validate the three sets of simulation points with varying architectural configurations, e.g., different interconnection network configurations. We present our results in Section V showing that the combination vector is the most accurate representation, i.e., it performs closest to running the applications to completion.

IV. PROFILING PROGRAM BEHAVIOR

As we described in the previous section, BBV is an attractive structure to represent the execution of a given interval. The BBV records the number of times a basic block is executed in the interval. On the other hand, the BBV element used in SimPoint is the number of times a basic block is executed multiplied by the number of instructions of that basic block. The purpose of doing this multiplication is to make sure that the summation of the elements equals to the length of the interval (the number of instructions executed within the interval). The number of basic blocks in one program usually ranges from several thousands to hundreds of thousands [16]. For the BBVs, the number of dimensions is the number of executed basic blocks. With so many dimensions, K-means, the clustering algorithm inside SimPoint, hits the so-called "curse of dimensionality". In addition, the runtime of SimPoint increases with the number of dimensions. SimPoint developers introduced the random linear projection technique to reduce the number of dimensions. However, this process may lose some profiling information. It is reported that a dimension number of 15 was found to be sufficient to distinguish the different phases of execution [16]. However, since we are simulating a 16-core CMP machine, it is complicated to gather this basic block information. Our modeling infrastructure is based on Simics, which complicates the extraction of basic block information. Compared to BBV, ICV is much easier to obtain and maintain. It only has 16 dimensions in our case, and performs similar to BBV. The main goal of our work is to show that incorporating the network traffic information will improve the accuracy of the phase analysis. Therefore, we use ICV to be the representation of the execution. We use the ICV on a 16-core CMP as our baseline to detect phases of execution and validate them. To collect the traffic count vectors, we put several counters in each router. In our evaluation, we assume a direct topology [27], where for each processor, there is a single router associated with the processor. In our experiments, each router is directly connected to a private L1 cache, shared L2 cache, and a memory directory. In the remainder of this paper, a "node" always denotes a core including a router with the L1 cache, L2 cache and directory connected to it. At the runtime, the counters calculate the number of packets generated by the node, destined for the node and also the intermediate packets going through the node separately. The element in the traffic count vector corresponds to the traffic triggered or responded or passing by the particular core. The execution interval of the TCVs has the same length with the ICVs. In this case, the summation of the elements in a traffic count vector will vary. However, since SimPoint first normalizes all the frequency vectors [10], this variation will not affect the extraction of different execution phases.

As for the combination approach, we can combine the two kinds of vectors with different weights. In our simulation, we assume that the communication structure is as important as instruction behavior, therefore in the combination, we give ICV and TCV roughly the same weights. We suppose that the weights determination should be related to the number of cores on chip and the complexity of the on-chip network. We believe that as the number of cores on chip scales up, the interconnect network traffic will have more influence on the execution, which means that TCV will become more important. Even in the combination vectors, the number of dimensions will not exceed 64 for our 16-core chip multiprocessor simulation. This also indicates that we will not lose too much information by random linear projection to reduce the number of dimension to 15.

V. EXPERIMENT SETUP

In our work, we first use GEMS [28] /SIMICS [29] to profile the target applications' behavior. We then use SimPoint [10] to detect phases based on those profiling information.

SIMICS is a functional full system simulator. We use SIMICS to simulate a Solaris-Sparc system. GEMS is an extension of SIMICS, which works together with SIMICS. GEMS can support chip multiprocessor simulations. It has two modules, which are all built on top of SIMICS. One is called Ruby and the other is Opal. The Ruby module implements a detailed memory system. Garnet [30], a detailed module of on-chip interconnection network, is also embedded into Ruby. In our work, we use Ruby module to simulate the memory hierarchy and the point-to-point on-chip network topology. The flexible network provided by Garnet is used to example the impact of different on-chip network architecture on detecting program phases. Table I describes the specifications of our simulated machine in GEMS.

We simulate a 16-core CMP system. Cache coherence is managed by a 2-level directory protocol for CMP. Two types of on-chip network topologies are simulated – a point-to-point (P-To-P) topology and a 2D mesh (4x4) topology. The point-to-point topology represents an "ideal" on-chip network topology as all 16 nodes on the chip are fully connected with zero router delay and no bandwidth limitation. However, since GEMS require a non-zero link latency, we introduce a single-cycle link latency in our simulation. As a result, there is no intermediate router delay as any node can be reached within a single hop. We also evaluate a 2D mesh network which is commonly used in many chip multiprocessors including the TRIPS processor [31], the 80-node Intel's Teraflops research chip [32], and the 64-node chip multiprocessor from Tilera [33]. We run 3 sets of simulations with different router delays. The router delays are assumed to be 1-cycle, 4-cycle and 8-cycle, respectively. We assume a dimension-ordered routing (DOR) where packets are routed first in the X-dimension and then, in the Y-dimension. The use of DOR simplifies the flow control as no additional virtual channels are needed to avoid routing deadlock.

SimPoint [16] is used for automatically characterizing program behavior. We use SimPoint to process the profiling information of the execution and then group the execution intervals into a number of clusters. CPI (cycles/instruction) and execution time are recorded for the selected simulation points for the further evaluation.

We run two sets of benchmarks: NU-MineBench [34] and SPLASH-2 [26]. All the applications are parallel programs. NU-MineBench is a data mining benchmark suite containing a mix of representative data mining applications from different application domains. We use HOP, K-means, and ScalParC from NU-MineBench, and Barnes, LU, Radix, and Water-Spatial from SPLASH-2 benchmarking suites. HOP is a density-based grouping application for clustering.

TABLE I. SIMULATED MACHINE CONFIGURATIONS

CPU	Sixteen 1GHz SPARCv9, 2-way, in-order
L1 Cache	4-way split, 64KB, 3-cycle latency
L2 Cache	4-way split, 16MB, 6-cycle latency
Memory	4GB
Directory	Split, 80-cycle latency
On-chip Network	P-To-P, 4x4 mesh topology, 1-cycle link latency
Router	1, 4, 8-cycle delay

TABLE II. SIMULATION PROPERTY OF THE TARGET APPLICATIONS

Application	Instructions [Million]	Packets	Intervals	Cycles [Million]
Barnes	9545	42774860	9545	12564
LU	949	714165	949	1072
Radix	1609	3155094	1609	2064
Water-Spatial	2617	1957989	2617	2845
HOP	219	1115777	219	406
K-means	2387	2417238	2387	2732
ScalParC	2419	48574193	2419	11261

Application	Packets/ Interval	Cycles/ Interval	Simulation Hours
Barnes	4481	1316291	72
LU	753	1129610	7
Radix	1961	1282784	12
Water-Spatial	748	1087122	16
HOP	5095	1853881	1
K-means	1013	1144532	27
ScalParC	20080	4655229	56

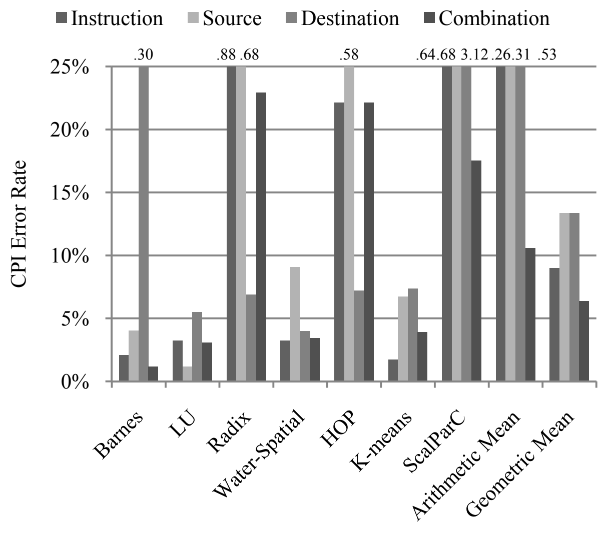


Figure 2. Error rate comparison of basic point-to-point on-chip network

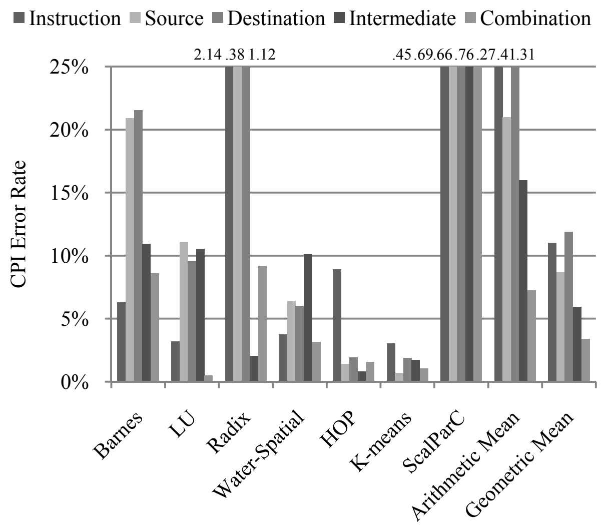


Figure 4. Error rate comparison of 4x4 mesh on-chip network with 4-cycle router delay

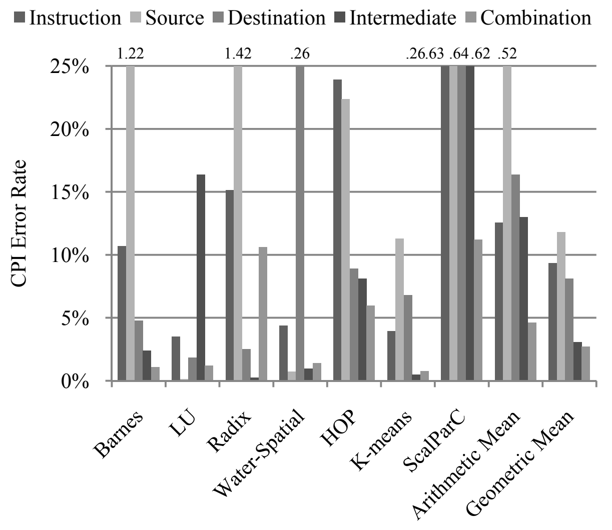


Figure 3. Error rate comparison of 4x4 mesh on-chip network with 1-cycle router delay

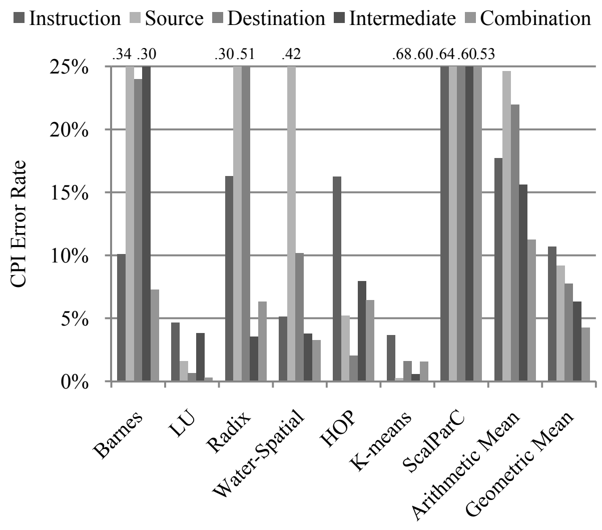


Figure 5. Error rate comparison of 4x4 mesh on-chip network with 8-cycle router delay

K-means is a well-known mean-based data partitioning application. ScalParC is a classification application based on decision tree classification. The SPLASH-2 benchmark suite contains two kinds of programs: complete applications and computational kernels. The Barnes application simulates the interaction of a system of bodies in three dimensions. The LU kernel factors a dense matrix into a lower triangular and an upper triangular matrix. Radix is an integer radix sort kernel. Water-Spatial is an extension of Water application in SPLASH. Table II describes the simulation properties of these target applications; it provides the number of instructions executed, simulation time (based on hours) of the whole execution, number of cycles simulated, total number of interconnect network traffic (total number of packets generated), number of intervals, and average number of packets per interval for the target applications. The simulation time is collected from a detailed architecture simulator. We can see that ScalParC, HOP and Barnes are the most traffic-intensive applications.

VI. RESULTS

In this section, we compare the performance of the three techniques in terms of CPI error rate and execution time.

Figure 2 shows the average CPI error rate of three approaches when compared to full application execution (i.e., running the applications to completion). Please note that many of the full application simulations take several days to complete and hence are impractical when a large number of architectural configurations are to be tested. In Figure 2, we show the CPI error rate of phases detected by 4 different profiling information. First one is phases detected by instruction profiling – ICV, we denote it as *Instruction* in the figure; second one is phases detected by the vector which records the number of packets generated by each node, we denote it as *Source*; the third one is phases detected by the vector which records the number of packets destined for each node and is denoted as *Destination*; the last one is phases detected by the combined vector which includes all of the above three kinds of profiling information and is denoted as *Combination*. On average, simulation points selected by ICV perform better than those selected by TCV in the ideal network and the performance of the combination approach is the best. We choose geometric mean to evaluate the performance of our proposed approach. Our experimental results show that there is large variation in error rates among different applications. We want to prevent one extreme case dominating the analysis of the trends. Therefore, we have used the geometric mean to draw general conclusions about the main trends in our approach. The average (geometric mean) CPI error rate of the combination technique is reduced by 29.0% compared with the error rate of the ICV technique (from 8.98% to 6.38%). In general, the error rate of the TCV technique is greater than that of the ICV technique, that is mainly because that P-To-P on-chip network is approximately an ideal network. In this ideal network, there are no bandwidth or buffer size limitations; so the transactions of interconnect network packets are fast. Since nodes are fully connected in this network, the interconnect packet can be delivered directly from the source node to the destination node. In this case, the interconnect traffic contributes little to the program execution time. Hence, the network traffic does not have significant influence on the CPI in this case. On the other hand, the instruction count vector is a better representative of the execution of the interval. While, even though the instruction behavior seems to dominate the program execution, for some applications, the CPI error rate of the ICV technique is high, e.g., for Radix and ScalParC, the error rate is over 25%. These applications generate relatively larger amount of interconnect packets. Consequently, the instruction behavior is not as dominant as other applications. In other words, in the ideal network, more interconnect traffic contributes to higher error rate of the ICV technique. For the applications generating more interconnect traffic, if we only consider the ICV to identify phases of program execution, this may result in improper clustering of the execution intervals for these applications. At last, we combine the ICV technique with the TCV technique. In this case, since the interconnect traffic contributes less than the instruction behavior, the combination vector contain big portion of ICV along with small portion of TCV. The ICV/TCV ratio vary from ~100 to ~10, this ratio depends on how many interconnect packets are generated during the program execution. For ScalParC, HOP, Barnes and Radix, the ICV/TCV ratio is relatively low. Overall, our combination technique provides better performance than the base line ICV technique: the arithmetic mean of the CPI error rate can be reduced by 59.8% (from 26.34% to 10.59%) and the geometric mean can be reduced by 29.0% (from 8.98% to 6.38%).

Even though we limited the number of simulation points to not exceed 10, the results show that for all the target applications the CPI error rate of the combination technique is less than 25%.

Figure 3 through Figure 5 show the results of the second on-chip network configuration using a 2D mesh topology, which is more representative of topologies used in CMP machines. Figure 3, 4, and 5 present the results when the router delay is set to 1-cycle, 4-cycles, and 8-cycles, respectively. In this set of experiments, the on-chip router delay is set to be a single cycle. In Figure 2, we show the CPI error rate of phases detected by 5 different profiling information. *Instruction*, *Source*, *Destination* and *Combination* have been explained in the former paragraph, *Intermediate* denotes the phases detected by the vector which records the number of packets passing through each node. In other words, the packets counted are neither generated by the node nor destined for the node, they just pass by. It is apparent that, in the former on-chip network (P-To-P), there is no intermediate packets, all the interconnect packets are transmitted directly from the source node to the destination node. However, in the 4x4 mesh network, since the nodes are not fully connected, an interconnect packet may traverse several nodes to get to the destination. Depending on the router delay and link latency, varying number of intermediate packets will be accumulated on all the nodes. The CPI error rate shows quite different trends compared to the basic network configuration. This 4x4 mesh network is much more realistic. In this case, the interconnect network traffic has a much more profound impact on the execution of the program. Contrary to the case in the P-to-P on-chip network, the TCV becomes a more important signature of the execution for the applications that generate large amounts of network packets. On the other hand, for the applications with relatively small amount of interconnect network traffic, TCV still remains a relatively less important representation of execution. We first compare the three kinds of TCV (Source, Destination and Intermediate) techniques: on average, Intermediate technique gives the best performance in terms of CPI error rate (11.80%, 8.10%, and 3.06%, respectively). These results indicate that in the more realistic on-chip network, the intermediate packets contributes more than the source packets, destination packets, and even the instruction behavior (9.35% CPI error rate) to the program execution. Therefore, in the combination technique, when we combine the ICV with the TCV, we only consider the intermediate packets, and the ICV/TCV ratio is set to be roughly 1:1. On average, the combination technique provides a 71.0% improvement (reducing the error rate from 9.35% to 2.71%) on the accuracy over the base ICV technique. In this configuration, LU performs worse with Intermediate technique applied; this is because the traffic generated by this application is relatively small which results in a small amount of intermediate packets. Therefore, the intermediate traffic may not be as dominant as the instruction behavior. Overall, the combination technique reduces the arithmetic mean of CPI error rate by 63.2% (from 12.57% to 4.62%)

and reduces the geometric mean of CPI error rate by 71.0% (from 9.35% to 2.71%) compared with the basic ICV technique.

In Figure 4 and 5, we apply 4-cycle and 8-cycle router delay, respectively. As the router delay is increased, the contribution of the intermediate packets becomes more important. Therefore, in order to increase execution accuracy, we decrease the ICV/TCV ratio furthermore in the combination vector. And the combination technique reduces the CPI error rate by 69.0% (from 11.01% to 3.41%) and 60.0% (from 10.70% to 4.27%), respectively.

We also examine the execution time of the simulation points based on the three phase detection techniques. Figure 6 through Figure 9 show the execution time of simulation points selected by the ICV, TCV, and combination-based phases. The results show that the TCV and the combination techniques are comparable in terms of their simulation time and in fact reduce the overall simulation time compared to the phases selected via the ICV technique. For the ideal network, the geometric mean of the simulation time of the points selected by the combination technique is 6.1% less than that of the basic ICV technique. When compared to the whole execution, the execution time of the simulation points generated by SimPoint can be reduced by over two orders of magnitude. For the 4x4 mesh network, the average execution time of the combination technique selected simulation points is 10.2%, 27.3% and 14.2% less than that of the ICV-

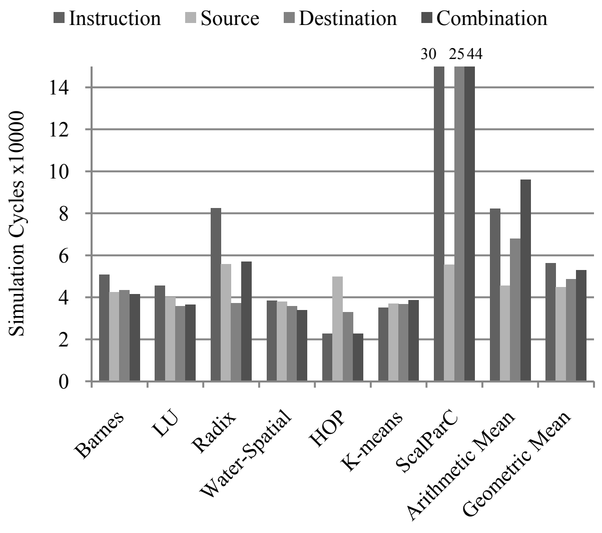


Figure 6. Execution time comparison of basic point-to-point on-chip network

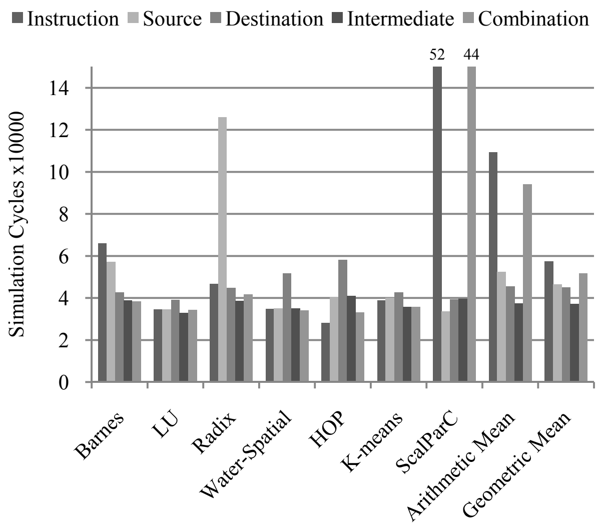


Figure 7. Execution time comparison of 4x4 mesh on-chip network with 1-cycle router delay

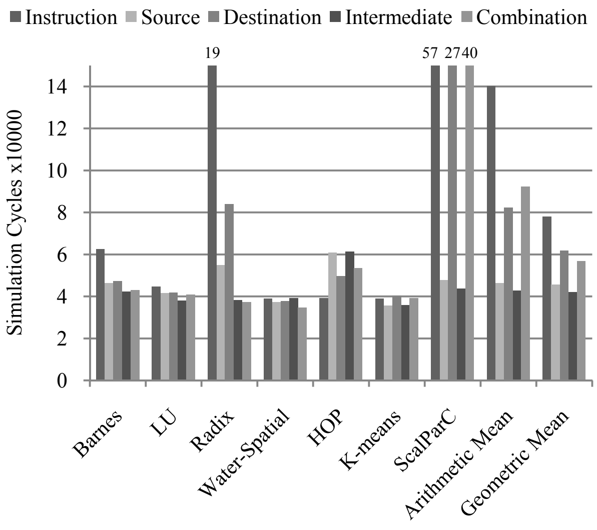


Figure 8. Execution time comparison of 4x4 mesh on-chip network with 4-cycle router delay

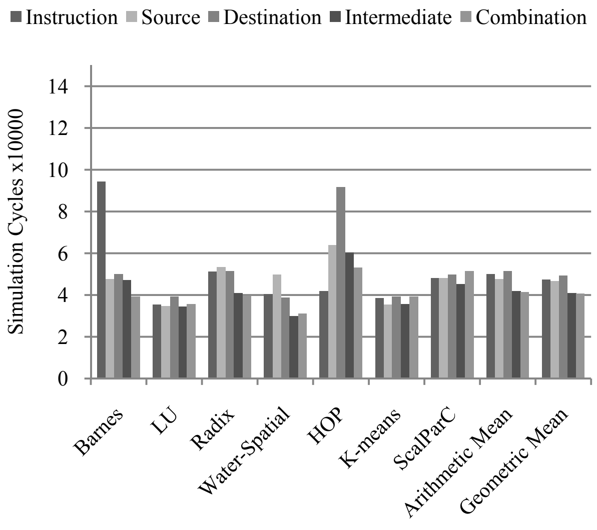


Figure 9. Execution time comparison of 4x4 mesh on-chip network with 8-cycle router delay

selected simulation points for 1-cycle, 4-cycle and 8-cycle router delays, respectively. As a result, on the whole, phase identification through the combination technique performs better in terms of accuracy without sacrificing performance in terms of execution time.

VII. Summary

In this paper, we presented the impact of on-chip network traffic patterns on phase identification of parallel programs running on CMP machines. We proposed new techniques to detect program phases. We first introduced a frequency vector called Traffic Count Vector (TCV) that collects information about the number of communication packets generated by each core during an interval. We also utilized a common technique called Instruction Count Vector (ICV), which counts the number of instructions executed on a core during each interval. In addition, we developed a hybrid scheme by combining the two vectors into a single vector. We evaluated the simulation points generated by these three frequency vectors. Our simulation results show that with the consideration of the on-chip network traffic pattern, higher accuracy (up to 71.0% less error rate when compared to the ICV based phase detection) can be achieved when only simulating the selected simulation points, without sacrificing performance while saving a significant amount (over 99%) of simulation time.

Acknowledgement

This work was supported in part by NSF grants CNS-0551639, IIS-0536994, CCF-0747201, CCF-0541337, and OCI-0724599; DoE CAREER Award DEFG02-05ER25691; and by Wissner-Slivka Chair funds. We would also like to thank anonymous reviewers for their insightful comments.

References

[1] Dhodapkar, A.S. and J.E. Smith. Comparing Program Phase Detection Techniques. in Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on. 2003.

[2] Smith, J.E. and A.S. Dhodapkar. Dynamic Microarchitecture Adaptation via Co-Designed Virtual Machines. in Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International. 2002.

[3] Dhodapkar, A.S. and J.E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. in Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on. 2002.

[4] Balasubramonian, R., et al. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. in Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on. 2000.

[5] Merten, M.C., et al., An Architectural Framework for Runtime Optimization. Computers, IEEE Transactions on, 2001. 50(6): p. 567-589.

[6] Barnes, R.D., et al. Vacuum Packing: Extracting Hardware-Detected Program Phases for Post-Link Optimization. in Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on. 2002.

[7] Biesbrouck, M.V., T. Sherwood, and B. Calder, A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation, in Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software. 2004, IEEE Computer Society.

[8] Tullsen, D.M., S.J. Eggers, and H.M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. in Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on. 1995.

[9] Sherwood, T., S. Sair, and B. Calder. Phase Tracking and Prediction. in Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on. 2003.

[10] Hamerly, G., et al., SimPoint 3.0: Faster and More Flexible Program Phase Analysis. Journal of Instruction-Level Parallelism, 2005. 7: p. 1-28.

[11] Annavaram, M., et al. The Fuzzy Correlation between Code and Performance Predictability. in Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on. 2004.

[12] Cho, C.-B. and T. Li, Complexity-Based Program Phase Analysis and Classification, in Proceedings of the 15th international conference on Parallel architectures and compilation techniques. 2006, ACM: Seattle, Washington, USA.

[13] Davies, B., et al., iPART : An Automated Phase Analysis and Recognition Tool. 2003, Microprocessor Research Labs - Intel Corporation.

[14] Isci, C. and M. Martonosi. Identifying Program Power Phase Behavior Using Power Vectors. in Workload Characterization, 2003. WWC-6. 2003 IEEE International Workshop on. 2003.

[15] Lau, J., et al. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. in Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on. 2005.

[16] Sherwood, T., et al., Automatically Characterizing Large Scale Program Behavior, in Proceedings of the 10th international conference on Architectural support for programming languages and operating systems. 2002, ACM: San Jose, California.

[17] Vandeputte, F. and L. Eeckhout, Phase Complexity Surfaces: Characterizing Time-Varying Program Behavior, in High Performance Embedded Architectures and Compilers. 2008, Springer Berlin / Heidelberg. p. 320-334.

[18] Sherwood, T. and B. Calder, Time Varying Behavior of Programs. 1999, UC San Diego.

[19] Sherwood, T., E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. in Proceeding of International Conference on Parallel Architectures and Compilation Techniques. 2001.

[20] Isci, C. and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. in Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on. 2003.

[21] Isci, C. and M. Martonosi. Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques. in High-Performance Computer Architecture, 2006. The Twelfth International Symposium on. 2006.

[22] Duesterwald, E., C. Cascaval, and D. Sandhya. Characterizing and Predicting Program Behavior and Its Variability. in Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on. 2003.

[23] Liu, W. and M.C. Huang, EXPERT: Expedited Simulation Exploiting Program Behavior Repetition, in

Proceedings of the 18th annual international conference on Supercomputing. 2004, ACM: Malo, France.

[24] Perelman, E., et al. Detecting Phases in Parallel Applications on Shared Memory Architectures. in Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. 2006.

[25] Hind, M.J., V.T. Rajan, and P.F. Sweeney, Phase Shift Detection: A Problem Classification. 2003, IBM Research Division.

[26] Woo, S.C., et al., The SPLASH–2 Programs: Characterization and Methodological Considerations, in Proceedings of the 22nd ISCA. 1995.

[27] Dally, W.J. and B. Towles, Principles and Practices of Interconnection Networks. 2004: Morgan Kaufmann.

[28] Martin, M.M., et al., Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset, in Computer Architecture News (CAN). 2005.

[29] Magnusson, P.S., et al., Simics: A full system simulation platform. Computer, 2002. 35(2): p. 50-58.

[30] Agarwal, N., L.-S. Peh, and N. Jha, Garnet: A Detailed Interconnection Network Model inside a Full-system Simulation Framework, in CE-P08-001, Dept. of Electrical Engineering, Princeton University. 2007.

[31] Gratz, P., et al., Implementation and Evaluation of On-Chip Network Architectures, in International Conference on Computer Design (ICCD). 2006.

[32] Vangal, S. and e. al., An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, in IEEE Int'l Solid-State Circuits Conf., Digest of Tech. Papers (ISSCC). 2007.

[33] Agarwal, A., et al., Tile Processor: Embedded Multicore for Networking and Multimedia. Hot Chips 19, Stanford, CA, 2007.

[34] Narayanan, R., et al., MineBench: A Benchmark Suite for Data Miningworkloads, in Proceedings of the IEEE International Symposium on Workload Characterization (IISWC). 2006.