Loop Block Profiling with Performance Prediction

Mohsin Khan[*1] , Maaz Ahmed[*2] , Waseem Ahmed[#3] , Rashid Mehmood[#4] , Abdullah Algarni[#5] , Aiiad Albeshri[#6] , Iyad Katib[#7]

[*]Dept. of Computer Science and Engineering, HKBK College of Engineering, Visvesvaraya Technological University, Bangalore, India

[#] Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, KSA

***Abstract*—*With increase in the complexity of High Performance Computing systems, the complexity of applications has increased as well. To achieve better performance by effectively exploiting parallelism from High Performance Computing architectures, we need to analyze/identify various parameters such as, the code hotspot (kernel), execution time, etc of the program. Statistics say that a program usually spends 90% of the time in executing less than 10% of the code. If we could optimize even some small portion of the 10% of the code that takes 90% of the execution time we have a high probability of getting better performance. So we must find the bottleneck, that is the part of the code which takes a long time to run which is usually called the hotspot. Profiling provides a solution to the question: which portions of the code should be optimized/parallelized, for achieving better performance. In this research work we develop a light-weight profiler that gives information about which portions of the code is the hotspot and estimates the maximum speedup that could be achieved, if the hotspot is parallelized.*

***Keywords*—*Profiling, Loop Block Profile, Code Analysis, Performance Prediction, Speedup Estimation*

I. Introduction

In the last decade we have seen a rapid growth of programming techniques towards parallelism. Processor clock frequencies are at their peak limits and latest processors have multiple cores, and accelerators have started to become more common on HPC systems, assisting to process the scientific programs faster. Hence parallelism is now important for all the legacy and newly developed applications. But parallel programming is not so easy than its sequential counterpart, and exploiting parallelism is much more difficult to achieve. Recently, there is an extensive use of parallel programs to build high performance applications. These applications run on different heterogeneous architectures.

With increase in the complexity of High Performance Computing systems, the complexity of applications has increased as well. Achieving better performance on leading technology systems is crucial. The inability to utilize such HPC systems efficiently causes a wastage of resources. It is observed that in order to achieve better performance by effectively exploiting parallelism from these architectures, we need to analyze/identify various parameters such as, the code hotspot (kernel), main memory usage, cache utilization, execution time, etc of the program. As a result, there is a real need of effective tools that can point out a variety of performance bottlenecks in scientific applications. Hence profiling plays a very important role in code optimization for both serial and parallel computing. Profiling provides a solution to the question: which portions of the code should be worked on, for achieving better performance. Also there is a need of a profiler that is fast and does not add a huge overhead on the execution of time of the program. Adding a huge overhead on the execution time will produce incorrect execution time there by giving inaccurate results. The profiler we propose here only performs small modification on the source code and is very light-weight in use, that is, it does not add huge overhead on the execution time of the input program.

The main contribution of this research paper are

- a light-weight loop block profiler that gives information about the hotspot loops of the code, such as, identify which loops of the code takes longer execution time, identifies the coverage of the hotspot loops.
- speedup prediction, if the bottleneck loops are parallelized

II. Background

Statistics say that a program usually spends 90% of the time in executing less than 10% of the code, also known as the 90/10 rule [1]. So if we can optimize the 90% of the code that takes 10% of the

execution time we will not get better performance. Rather if we optimize even some small portion of the 10% of the code that takes 90% of the execution time we have a high probability of getting better performance. So, we must find the bottleneck, that is, the part of the code which takes a long time to run which is usually called the hot-spot /hot purpose. We can get this information by profiling code i.e., by use of profilers. Once we identify the hot-spot we can try to optimize it and/or parallelize it. And also try to run the code on parallel architectures to get good performance. (Minimize the execution time).

Most kernels/hot-spots are located inside loops and much of the parallelism is found inside the loops. Hence it is very crucial to identify bottleneck loops.

A. Speedup Estimation

To estimate the speedup that can be achieved after parallelization there are many models, out of which the most commonly used is Amdahl's law [2] as described in equation 1. Amdahl's law gives the maximum theoretical speedup that could be achieved, that is, it only gives an upper bound of the speedup that can be achieved.

$$S = \frac{N}{1 + (N-1)\alpha} \tag{1}$$

S = Theoretical Speedup
N = No of processors
α = Percentage of the serial portion of the program (range: 0 to 1)

B. Applications

Source code analysis has become important task in the compiler community and other software engineering domains [3]. This section lists some of the applications of source code analysis.

- comprehension [4]
- debugging [5]
- optimization techniques in software engineering [6]
- performance analysis [7]
- reverse engineering [8]
- program evolution [9]
- testing [10]
- visualization of analysis results [11]

III. Methodology

The profiler takes as input a configuration file which contains information about the input program for which the profiling has to be done. This file contains the name and location of the input program's source file(s), the compile command and the execution command for the input program. The profiler produces as output the profiling information about the input program. More specifically the profile contains the timing information of loops, the speedup that can be achieved if this loop is parallelized. The loops are ranked based on the time they spend in execution, with the lowest rank as most time consuming loop. The input program source file passes through many stages. These stages are initialization, instrumentation, compilation & execution and speedup prediction & profile generation. These stages are described in the following sections.

A. Initialization

The profiler first parses the input program into an Intermediate Representation (IR). A good IR is one that is independent of the source and target languages. IR allows to divide the difficult problem of translation into simpler, more manageable pieces. In this research, IR is used in order to extract various details of the code and also to maintain syntactic correctness of the code at later stages as well. Here all the functions (sub-routines) used in the input program are identified, this is done in order to parse the source code functions (sub-routines) one by one, iteratively . The other information from the input configuration file is also stored for later use.

B. Instrumentation

After parsing of the input program the source code functions are iterated one by one. In each function the loop blocks are identified. Only the outer loops are profiled since profiling of inner loops will give partial information and profile generated with this information will not be very concise. After the loop blocks are identified they are marked. The instrumentation code is initialized and infused around the blocks that are marked. This instrumented source code obtained after instrumentation, is saved in a separate file so that the source file(s) are not modified.

C. Compilation & Execution

The instrumented code file is now compiled by taking the earlier saved compile command. The compile and execution command is altered by adding necessary flags and libraries that are required for the instrumentation code. The instrumented code is then compiled and executed. If any errors in compilation or execution, it is reported in the error log.

D. Speedup prediction & profile generation

To predict the speedup Amdahl's law mentioned above, is used. To calculate the speedup the number of processor cores (N) is required. This is taken as input parameter in the configuration file. To use Amdahl's law the value of α is also required. It is calculated by using the timing information and

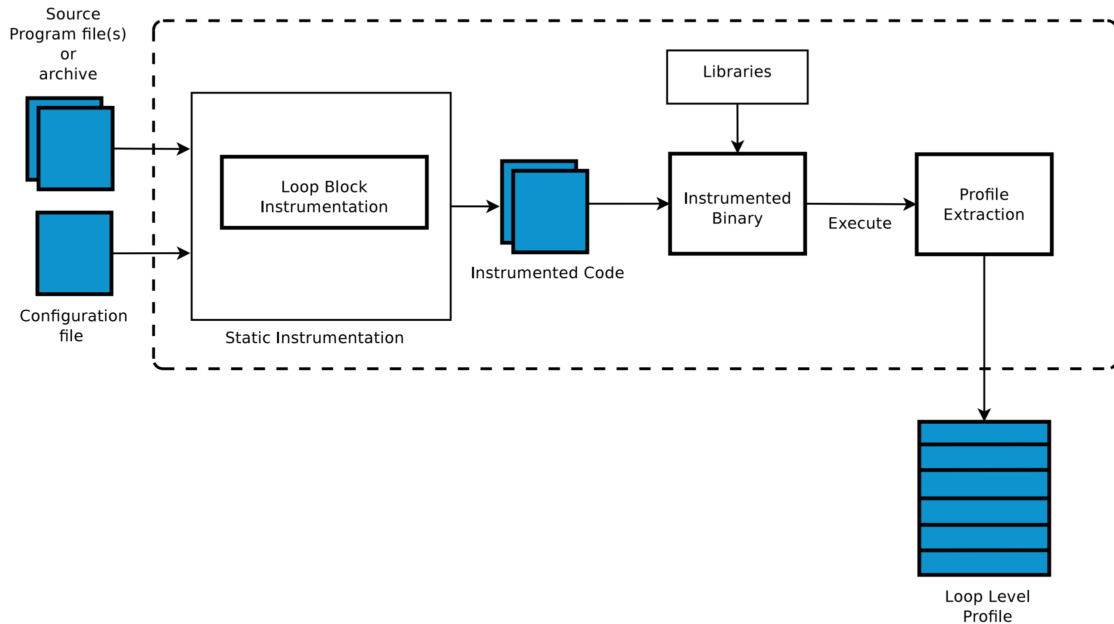

Figure 1. Overview of Profiler System Architecture

applying it in equation 2. Now α and n are used in Amdahl's law for speedup prediction for each loop block.

$$\alpha = 1 - \left(\frac{exec\, time\, of\, the\, loop}{total\, exec\, time\, of\, the\, prog} \right) \tag{2}$$

The output generated while execution of the instrumented code is captured as JavaScript Object Notation (JSON) format [12]. The JSON is then parsed to extract data such as, function name in which the loop is present, block number, execution time of the loop, coverage percentage (ratio of execution time of the loop block and total execution time of the program) and speedup if the loop block is parallelized. All this is combined and saved into .csv file for future reference.

IV. Experimental Testbed

The experiments were carried out on a High performance computing (heterogeneous) server with configurations as shown in table I.

A. Server

INTEL XEON E5-2620: This processor is built on the Ivy Bridge Micro-architecture. It is a two socket hexa core processor. Each core has 2 threads which runs at 2 GHz and contains 32KB L1 instruction and data cache and has L2 cache of the size 256 KB. The total size of the L3 cache is 15 MB (shared). The size and type of the ram is 32GB and DDR3 respectively.

Table I
Configuration of the systems

	Server_1	
System	Intel(R) Xeon(R) CPU E5-2620	NVIDIA Tesla K40c
Micro architecture	Ivy Bridge Micro architecture	Kepler Micro architecture
Sockets	2	1
Cores per socket	6	2880
Threads per socket	12	-
Core speed	2.00 GHz	745 MHz
Core size	32 nm	28 nm
Ram type	DDR3	GDDR5
Ram size	32 GB	12 GB
Memory bandwidth	42.6 GB/s	288 GB/s
L1 Data Cache	6 x 32K	-
L1 instruction Cache	6 x 32K	-
L2 Cache	6 x 256K	-
L3 Cache	15360K	-
OS	OpenSUSE	-
Compiler	GCC 4.8.5 / nvcc	-

NVIDIA Tesla K40c: This graphics card has 2880 cores, which has a frequency of 745 MHz. It's memory size is 12 GB and is of type GDDR5. It has a memory bandwidth of 288 GBps. The PCI Express 3.0 x 16 interface is used to connect this graphics card with rest of the system.

V. RESULTS AND ANALYSIS

The Loop Block Profiler was evaluated with 14 programs from the Polybench Benchmark suite [13] and Rodinia Benchmark suite [14]. For parsing the input program source code Pycparser [15] was used. The Profiler was run on the unmodified serial versions of the benchmarks to generate a Loop level profile for each program. The profiles generated were combined and a concise profile of few benchmark programs is shown in table II. As noticed, the profile consists of various fields - In_Function: shows to which function the Loop Block belongs to, Block_No: is the block number of the loop in the function, Speedup (if parallelized): is the predicted maximum speedup that could be achieved if the Loop Block is parallelized, Coverage %: is the portion of total time that the loop takes for execution, Time: Execution time of the loop or function, Function_Name: is the name of the profiled function.

It is observed that some benchmark programs have fine grained parallelism and some have coarse grained parallelism. This is indicated by the profiler. Benchmarks- *atax, bicg, gesummv, lu* have fine grained parallelism. To achieve high performance, loop blocks that have coverages greater than 20% must all be parallelized. Benchmarks programs- *gemm, syr2k, cholesky* are coarse grained parallelism programs where huge parallelism exists in only some portion of the program. To achieve better performance only loops with coverage of 75% or higher are needed to be parallelized.

The last column of table II shows the speedup that is obtained on parallelization of the profiled Loop Blocks. Each Benchmark's kernel was manually parallelized to increase the performance of the Benchmark. The speedup obtained by manual parallelization is within the limit that was estimated by the profiler (refer column 4 of table II). Hence the profiler accurately gives the upper bound (max) speedup that can be achieved after parallelization.

It was observed that the Loop Block profiler only added a small overhead. On evaluation of 10 Benchmarks for overhead, it was observed that the overhead incurred was less than 1%, on an average, of the execution time, with the lowest overhead of 0.1% and highest overhead of 2.1%. This overhead is very much negligible. The table III shows this overhead for Polybench Benchmark Suite.

It was seen that the Loop Block profiler was able to profile all the loop blocks that were executed during runtime of the benchmark execution. Table IV portrays this. It can be observed that the total number of for Loop Blocks in the source code are different (more in number) than the Loop Blocks Profiled. This is due to the fact that at runtime not all the functions (sub-routines) were called, and these uncalled functions contain some Loop Blocks, that is, the difference in the Total number and number of Loop Blocks profiled, is number of the loop blocks that are not executed at runtime. Suppose if these uncalled functions would have also been called (executed during runtime) then the Total number of Profiled Loop Blocks will be the same as total number of For Loop Blocks in the Benchmark source code.

VI. RELATED WORK

Profilers play an important role in software/hardware design, optimization, and verification. Different approaches have been proposed to implement profilers. The Kremlin tool [16] implements a practical oracle that predicts outcomes for sequential-code parallelization. The tool takes in an unmodified serial program and a few representative inputs, and outputs an ordered list of the regions that are likely to be the most productive for the user to parallelize. The approach used in [4] by Dubach et al. is to improve the performance of compiled code by searching the space of compiler options that control optimization levels in GCC. The main objectives considered in this paper are time taken for compilation and quality of the code. In [17] the authors describe the sampling approach to profile sequential programs. this approach counts procedure executions. In [18] Kim et al. proposed Parallel Prophet which implements an emulator along with a memory performance model to estimate speedup for the annotated regions of code. It assumes that the DRAM accesses do not vary when going from serial to parallel. This assumption is not always true. Parallel Prophet reduces the analysis overheads and takes into account memory bandwidth saturation, the approach is applicable to only offline decisions. Praun et al. [19] proposed dependence density metric to state the probability that two random tasks would have a dependency. This work is very closely related to Thread-level Speculative parallelization (TLS). In [20] Wu et al. proposed DProf which identifies may-dependencies and then find out the probability that these dependencies will occur using a compiler. Ketterlin and Claus [21] developed a tool named Parwiz which facilitates the programmer to find out potential parallelism by profiling the dependencies of the input program. In [22] Gao et al. introduce a source code cross profiler TotalProf, which measures the performance of an application.

All though there are a lot of profilers available, there is a need for a light-weight profiler that does not modify the source code much, there by not increasing the execution time of the program, and is extremely fast. The profiler we propose here only performs small modification on the source code and is very light-weight in use, that is, it does not add

Table II
PROFILED OUTPUT OF POLYBENCH BENCHMARK

Estimated Speedup shown is for N=4

Benchmark	In_Function	Block_No	Speedup (if parallelized)	Coverage %	Time	Speedup obtained manually
atax	init_array	4	1.71	55.46	0.04614322	
	kernel_atax	4	1.42	39.55	0.03290431	1.02
	init_array	0	1.00	0.03	0.00002288	
	kernel_atax	0	1.00	0.01	0.00001125	
bicg	init_array	4	1.74	56.67	0.05361487	
	kernel_bicg	4	1.44	40.56	0.03837169	1.05
	init_array	0	1.00	0.05	0.00004953	
	kernel_bicg2	0	1.00	0.01	0.00001178	
gemm	kernel_gemm	0	3.92	99.31	7.20921818	1.63
	init_array	0	1.00	0.26	0.01852181	
	init_array	4	1.00	0.20	0.01446096	
	init_array	8	1.00	0.19	0.01410919	
syr2k	kernel_syr2k	0	3.97	99.75	17.81511442	3.94
	init_array	0	1.00	0.16	0.02857401	
	init_array	4	1.00	0.08	0.01392111	
cholesky	init_array	8	3.30	92.98	78.02921866	
	kernel_cholesky	0	1.05	6.94	5.82375567	1.05
	init_array	0	1.00	0.04	0.03000305	
	init_array	4	1.00	0.02	0.01794518	
	init_array	12	1.00	0.01	0.01249174	
gesummv	init_array	0	2.31	75.70	0.03906825	
	kernel_gesummv	0	1.18	21.22	0.01095341	1.15
lu	init_array	8	2.24	74.03	78.13759824	
	kernel_lu	0	1.24	25.90	27.33702285	1.23
	init_array	0	1.00	0.03	0.03656997	
	init_array	4	1.00	0.01	0.01741910	
	init_array	12	1.00	0.01	0.01191741	

Table III
PROFILER OVERHEAD

Benchmark	Execution Time	Execution Time with Profiling	Overhead %
gemm	7.249	7.269	0.28
gesummv	0.058	0.057	1.72
bicg	0.091	0.093	2.20
mvt	0.053	0.054	1.89
syr2k	17.749	17.825	0.43
atax	0.080	0.082	1.62
cholesky	82.902	83.398	0.60
gramschmidt	31.836	31.884	0.15
lu	105.950	106.078	0.12
gemver	0.132	0.133	0.76
		Average	**0.98** %

Table IV
PROFILER OVERHEAD

Benchmark	Total For Loop Blocks (Outer)	Total no. of Loop Blocks executed during Runtime	Total no. of Loop Blocks Profiled
gemm	5	4	4
gesummv	3	2	2
bicg	6	4	4
mvt	5	3	3
syr2k	4	3	3
atax	5	4	4
cholesky	5	4	4
gramschmidt	5	3	3
lu	5	4	4
gemver	6	5	5

huge overhead on the execution time of the input program. It was measured that the proposed profiler only added a small overhead that was less than 1% of the execution time, which is very negligible.

VII. CONCLUSION AND FUTURE WORK

We have developed a light-weight profiler that helps in capturing the profile of Loop Blocks of serial programs. The profile helps in identifying the hotspot loops and suggests loops that should be the focused upon to exploit parallelism. This profile helps the user to avoid parallelization of other loops that do not have much significance and parallelization of these of loops will not increase the performance of the program. The profile also gives a brief idea to the user about the speedup that can be achieved after parallelizing the loops. This allows the programmer to decide whether he should proceed further in trying to optimize the program and/or running the program on a HPC system, in case the HPC system is not present. The profiler can be used with ease without any prior planning by the user. The profiler adds very less execution overhead to the program that is profiled.

In future work, we will focus on adding data dependence analysis to the profiler. We intend to add a new and accurate mathematical model for speedup prediction. We also intend to add identification of

energy consumption of Loop blocks in our next work.

ACKNOWLEDGEMENT

The authors would like to thank the CUDA Center of Excellence at IIT Bombay, India for providing access to their HPC system.

REFERENCES

[1] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," in *ACM SIGPLAN Notices*, vol. 38, pp. 189–198, ACM, 2003.

[2] D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[3] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering, 2007. FOSE'07*, pp. 104–119, IEEE, 2007.

[4] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th international conference on Computing frontiers*, pp. 131–142, ACM, 2007.

[5] H. Nilsson and P. Fritzson, "Lazy algorithmic debugging: Ideas for practical implementation," *Automated and Algorithmic Debugging*, pp. 117–134, 1993.

[6] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*, pp. 342–357, IEEE Computer Society, 2007.

[7] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering, 2007. FOSE'07*, pp. 171–187, IEEE, 2007.

[8] G. CanforaHarman and M. Di Penta, "New frontiers of reverse engineering," in *2007 Future of Software Engineering*, pp. 326–341, IEEE Computer Society, 2007.

[9] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, ACM, 2000.

[10] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*, pp. 85–103, IEEE Computer Society, 2007.

[11] D. Binkley and M. Harman, "Analysis and visualization of predicate dependence on formal parameters and global variables," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 715–735, 2004.

[12] D. Crockford, "Json: Javascript object notation," *URL http://www. json. org [Accessed: May 2017]*, 2006.

[13] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite (2011)," *URL http://www-roc. inria. fr/˜ pouchet/software/polybench*, 2015.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[15] E. Bendersky, "Pycparse (2010)," *URl: https://github. com/eliben/pycparser [Accessed: May 2017]*.

[16] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor, "The kremlin oracle for sequential code parallelization," *IEEE Micro*, vol. 32, no. 4, pp. 42–53, 2012.

[17] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, pp. 120–126, ACM, 1982.

[18] M. Kim, P. Kumar, H. Kim, and B. Brett, "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1318–1329, IEEE, 2012.

[19] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 185–196, ACM, 2008.

[20] P. Wu, A. Kejariwal, and C. Caşcaval, "Compiler-driven dependence profiling to guide program parallelization," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 232–248, Springer, 2008.

[21] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 437–448, IEEE, 2012.

[22] L. Gao, J. Huang, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, "Totalprof: a fast and accurate retargetable source code profiler," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 305–314, ACM, 2009.