Automatic Sequential to Parallel Code Conversion

The S2P Tool and Performance Analysis

Aditi Athavale, Priti Ranadive, M. N. Babu, Prasad Pawar, Sudhakar Sah, Vinay Vaidya, Chaitanya Rajguru
CREST, KPIT Cummins Infosystems Ltd.,
Pune, India

{Aditi.Athavale,Priti.Ranadive,M.babu,Prasad.Pawar,Sudhakar.Sah,Vinay.Vaidya,Chaitanya.Rajguru}@kpitcummins.com

***Abstract*— The way software programs are being written has been redefined since the introduction of multicore processors. Software developers have started writing parallel programs that are robust and scalable. This would ensure use of processor power being made available in the form of multiple cores. Though this trend is increasing, there are legacy applications that have been developed over the past few decades. Most of these applications are inherently sequential making no use of multithreading or parallel programming. If such applications are ported to execute on the multicore hardware as they are then optimal usage of all cores is not guaranteed. Such applications would ideally utilize only one core and the other cores would remain idle, unless the operating system supports some parallelism while scheduling. Hence there is a need to convert such legacy sequential codes to their parallel versions so that multicore hardware is exploited to the fullest. In this paper we present a tool that we have developed to automatically convert a sequential C code to parallel code. This Sequential to Parallel (S2P) tool is still in the development phase. We also discuss other parallelization tools available today, compare such tools with S2P tool and present our performance analysis results on different kind of multicore hardware.**

Keywords-automatic parallelization; sequential to parallel code conversion; S2P tool; multicore programming

I. INTRODUCTION

Parallel computing has been around for a few decades now but its applications were mainly found in the scientific computing domain. This is mainly because such applications involve mathematical computations and require huge computing power. Since individual computers are unable to provide such high computing power multiple computers were connected together to form clusters and grids. Scientific applications used these grids to perform computations that were independent of each other and later collate the results. This kind of computing is known as distributed computing.

As the demand for computing power for desktop applications increased, the desktop computers also became faster. The increase in clock frequencies results in increased heat dissipation and increased power consumption. To limit the heat dissipation and power consumption multicore processors were invented. It would not be an exaggeration to say that multicores processors give the power of distributed computing on a single chip. This paradigm shift in the hardware architecture has forced the software industry to reconsider the way software is written.

It has become increasingly important to write parallel programs so as to exploit available multicore hardware. Future processors will see increasing number of cores per chip and hence writing scalable programs is also a must along with error free parallel programs. There is also a need to convert legacy sequential programs to parallel programs so that they can exploit multiple cores after porting.

Parallelizing a sequential application demands for a huge investment of time and money since it involves understanding the application domain, understanding the application, identifying data and control dependencies and then actually writing and synchronizing various sections that would execute in parallel. There are various options like OpenMP, MPI, and CUDA that can be used to write parallel code. These tools would help in 'how-to' part of parallelization but do not help in 'what-to' parallelize decisions. Parallelization also needs expertise in parallel programming domain, to choose appropriate tool for the application in hand. Doing all this manually for a huge application is a humungous task. We started developing an automated parallelizing tool to address the problem of porting legacy applications to multicore hardware.

Automatic parallelization ensures that the programmer does not need to identify sections of code that are possible candidates for parallel execution. Programmer does not need to perform data dependency analysis to keep the program correctness intact. The programmer also does not need to insert parallel code or directives manually at relevant places. In addition to all these reduction of efforts automatic parallelization can sometimes result in shorter execution times on SMP and HT-enabled systems. Not all application parallelization result in shorter execution time or better performance. This depends on the inherit nature of the applications, how much parallelization it offers and the amount of overhead we are creating by parallelization. Our experimental results show different kinds of parallelization strategies used and the performance analysis results for different applications on SMPs and HT enabled machines.

Additionally, even after investing huge time and money for manual parallelization efforts the performance results are not guaranteed. Hence an automatic parallelization tool would be useful in such cases to know quick performance results for a given applications.

The rest of the paper is organized as follows: Section II gives a brief literature survey of the various tools available

DOI: 10.5176_2010-2283_1.4.113

for parallelization and compares them with the S2P tool. Section III gives details about the implementation of the S2P tool. In section IV we discuss the performance analysis results of the parallelized code generated by the S2P tool and also compare our results with results obtained using other tools.

II. LITERATURE SURVEY

There are several tools available for parallelization. In this paper we only mention tools that are automatic parallelization tools. To the best of our knowledge, following fully automated parallelization tools are available: Cetus, par4all, Intel C++ compiler and SUIF. There are several other tools, frameworks, and language extensions that are available but cannot be classified as fully automated and hence are not mentioned in this paper.

A. *The SUIF compiler*

The SUIF compiler [1] was the first of its kind. It was developed to automatically convert sequential dense matrix computations, written in C or FORTRAN, to parallel code for machines with shared memory. The compiler included various optimizations and passes for performing program analysis including symbolic analysis, parallelism and locality analysis, communication and synchronization analysis and code generation.

B. *The Intel Compilers*

The Intel compilers generate multithreaded code automatically. They target parallelization of applications where most of the computations are carried out by loops [2]. The parallelization of loops is based on the results of dataflow analysis in loops. The Intel compilers parallelize codes written in C, C++ and FORTRAN languages.

C. *The Par4All Tool*

Par4All is an automatic parallelizing and optimizing compiler for programs written in C and FORTRAN. It is based on PIPS (Parallelization Infrastructure for Parallel Systems) [3] source-to-source compiler framework. The 'p4a' is the basic script interface to produce parallel code from user sources. It takes C or FORTRAN source files and generates OpenMP or CUDA [4] output to run on shared memory multicore processor or GPGPU respectively.

D. *The Cetus Tool*

Cetus is a tool that performs source-to-source transformation of software programs, which are written in C language. It also provides basic infrastructure to write automatic parallelization tools or compilers. The basic parallelizing techniques Cetus currently implements are privatization, reduction variables recognition and induction variable substitution. Cetus enables automatic parallelization by using data dependence analysis with the Banerjee-Wolfe inequalities [6], array and scalar privatization.

After looking at these major automatic parallelization tools, we can easily understand that these tools focus on data parallelism in the programs. Most of the times, the data parallelism is exploited by parallelizing loops in the programs. However, parallelizable loops constitute only a small portion of programs. The question is can automatic parallelization tools target tasks level parallelism as well. Inclusion of task parallelization increases the reach of the parallelizing tools, so that these tools can support larger set of applications. The tool that is being discussed in this paper, the S2P tool, focuses on task as well as loop parallelization in the programs.

III. THE S2P TOOL

The S2P tool is an automatic parallelization tool that converts a sequential C source code to a parallel code. The parallel code is a multithreaded code with pthread and OpenMP constructs inserted at relevant places. Pthreads are used for task parallelization and OpenMP is primarily used for loop parallelization. We have also performed some experiments in which OpenMP's 'tasks' constructs are used instead of Pthreads, for task parallelization. Results of all the experiments are discussed in later sections.

Fig. 1 shows where S2P tool fits in a typical software execution model. S2P tool is a source to source conversion tool. Hence the parallel code, generated by the S2P tool, needs to be compiled like a sequential C code.

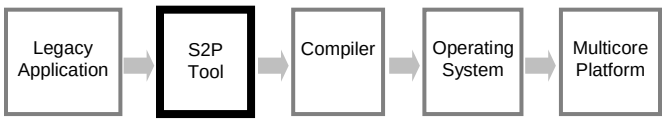


Figure 1: S2P tool in software execution model

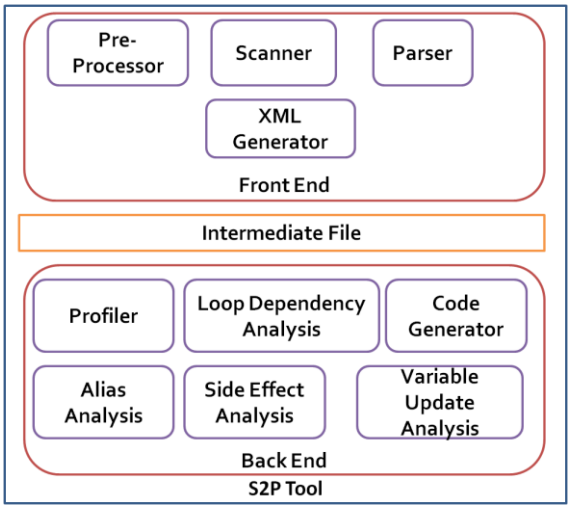


Figure 2: Block diagram of S2P tool.

The S2P tool consists of a front end that can scan and parse application code and an intelligent backend that performs static dependency analysis to identify parallelizable sections of code. The tool also consists of a code generator

that automatically generates parallel code. This parallel code is functionally similar to the sequential code that can execute faster than the sequential code and also that can optimally utilize all the available cores on the hardware. Fig. 2 shows high level block diagram of the S2P tool. In the following text, all the sub-modules of front end and back end are described in detail.

The front end contains pre-processor, scanner, parser and intermediate file generator. The pre-processor replaces the constants, which are present in terms of pragmas, and resolves the dependencies in header files. It then formats the sequential C code as required by the scanner. The scanner forms lexical tokens of the source code. The parser parses these tokens as per the ANSI C grammar and generates an intermediate file as output. This intermediate file contains all the code information that is subsequently used by different back end modules. The file stores metadata about each and every program constructs that is present in the sequential C programs. For a variable, information of its definition, data type and scope of the variable, files in which the variable is declared and accessed, lines on which the variable is used etc is stored. For a function, information of its definition, return types, other functions that call and get called by this function, parameters, arguments, and so on. Similar information is stored for iterative, control and selection statements as well.

The intermediate file is the input to the back end that performs static analysis for identifying dependencies among different sections of code. It transforms the code into pthread and OpenMP mixed code. The backend sub-modules include the profiler, blocks identifier, pointer alias analyzer, side effect analyzer, task dependency matrix (TDM) creator and code generator.

Blocks of code are identified based on logical scopes. Program constructs which are categorized as blocks are loops, function call sites, an 'if' statement etc. We have considered treating 'then' and 'else' blocks as separate blocks as well as the whole 'if-then-else' block as a single block. This decision is based on the profiling data obtained for the 'then' and 'else' parts. If the 'then' and 'else' parts execution times are greater than a threshold value and are comparable with each other they are treated as separate blocks. For example, if the 'then' block executes for 100 seconds and the 'else' block executes for 120 seconds they are considered separately. But if the 'then' block executes for 10 seconds and the 'else' part executes for 100 seconds the timings are not comparable from the scheduling perspective and hence we treat them are a single block.

Currently we have limited the block granularity to the outermost level of logical scope. For e.g. if there is an 'if-then-else' block inside a loop, we consider the loop as a block. Thus as of now we have considered only outer level logical scopes within the 'main()' function in a C code as blocks. In addition to these blocks, loops are treated separately for parallelization.

The S2P tool executes the sequential code off-line to get profiling information. This information is stored in an intermediate memory structure for further reference. The profiling information is important to take decisions about parallelization. For example, if a code block is found to be parallelizable but the execution time for that is below a certain threshold, then it will not be parallelized. The threshold is calculated considering the OpenMP and pthread thread creation and synchronization overheads. Thus blocks that have significant execution time with respect to the threshold value, are parallelized to ensure that the reduction in execution time is greater than the overhead of parallelization constructs. We also plan to implement block merging or splitting (changing granularity) based on the profiling information to obtain better performance results. This work is still in progress.

Pointers are majorly used in C programs and it is important to address pointer aliasing while analyzing dependencies among various blocks of codes that use pointers. We have implemented a flow insensitive and context insensitive inter-procedural approach for computing the aliases [7]. This approach was chosen for two major reasons – First and foremost important reason is that it is a safe approach. In case of even a slightest ambiguity, two variables are considered to be aliased. This ensures that the code is functionally correct and wrong parallelization decisions are not taken. However, it also implies that the parallelization performance may get reduced. Second reason is that if we want more precise information about aliases, the cost of implementation in terms of time, memory and complexity would be high as compared to the performances gained.

The side-effects of functions calls also need to be considered for getting correct dependency information among blocks of code. We have implemented the side-effect analysis algorithm as described in [8] with few modifications to accommodate exits, jumps and I/O related function calls in C programs like printf, fprintf, exit, etc. We have also implemented a simple method to detect the modification of variables [5]. This information is also useful to identify dependencies. The dependency information generated by the alias analysis module, side-effects analysis modules and variables update analysis is processed together and stored in a matrix, known as the Task Dependency Matrix (TDM) [9]. The rows and columns in the TDM represent tasks. In the current implementation of S2P tool, the tasks are equivalent to blocks. The cells in the TDM represent the dependencies between these tasks. Thus if n blocks are identified in the sequential code, n X n TDM is created. If there are no dependencies among blocks i and j then the ij^{th} element of the matrix is empty, whereas if the i^{th} block is dependent on the j^{th} block then the ijth element in the matrix contains the line number at which the dependency is identified. This dependency information is later on used by the code

generator module to insert synchronization constructs at the appropriate places.

All of these sub-modules present in the backend help track the task level parallelization in the program. To further increase the effectiveness of parallelization, loops are also analysed to exploit data parallelism in the program. In order to parallelize loops, the iterations of the loop should be able to run independently. In other words, two or more iterations of the loop should not access the same data location. Data accessed inside the loop nest is classified as variables that are arrays and others. For non-array type of variables, same data dependency analysis techniques, which are mentioned in the above sections, are used. For array type of variables, two dependency tests are used – Greatest Common Divisor (GCD) and Single Variable Per Constraint (SVPC) is used [10]. Both of these tests, check the dependencies of array locations based on the arithmetic of array indices and loop variants. Based on this analysis, if all the iterations are found to be independent, OpenMP constructs are used to parallelize the loop. All the data required for generating OpenMP constructs are derived and inserted in the desired format in a completely automatic way.

Once dependency results of all the above modules are generated, the code generator module inserts Pthread APIs and OpenMP constructs in the code. Broadly, tasks are parallelized by using Pthreads, and loops are parallelized by using OpenMP constructs. However, OpenMP 'task' constructs are also used to enable task level parallelism in the code, instead of Pthreads. In case of Pthreads, the following kinds of statements are inserted at appropriate places in the sequential code:

1. Thread creation constructs
2. Thread synchronization constructs
3. Thread exit constructs.

In case of OpenMP, loop specific and task specific constructs are inserted. When the parallelization constructs are inserted in the sequential code, the scheduling of the threads is left to the underlying operating system. S2P tool does not interfere in the scheduling of threads.

In some cases, even after parallelizing the program using S2P tool, it is possible to have few cores still lying idle. In order to further increase the performance by utilizing these idle cores, a new technique of *Induced Parallelization* has been developed. In this technique [11], 'then' and 'else' blocks of are checked for dependencies against each other. In case of common data access by both the blocks, local copies of data are created inside each section. Both the blocks are put inside Pthreads are allowed to run simultaneously on the idle cores. In order to this, the dependencies of these blocks are also checked with that of the preceding sections of 'if-then-else' block. The simultaneous execution of 'then' and 'else' blocks is done ahead of time. When the actual condition of 'if' statement is hit, only one of the 'then' and 'else' block is allowed to continue execution. Figure 3 gives a pictorial view of Induced Parallelism technique.

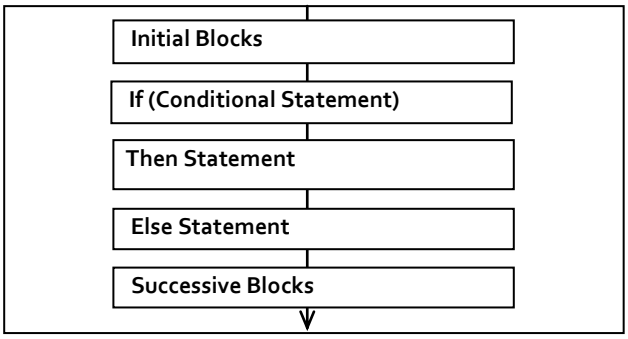


Figure 3: Block diagram of S2P tool

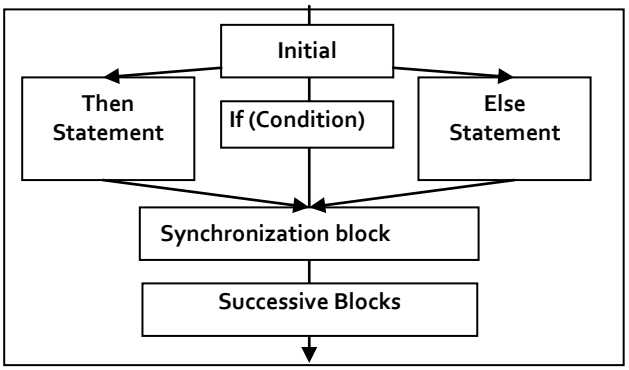


Figure 4: Induced Parallelism

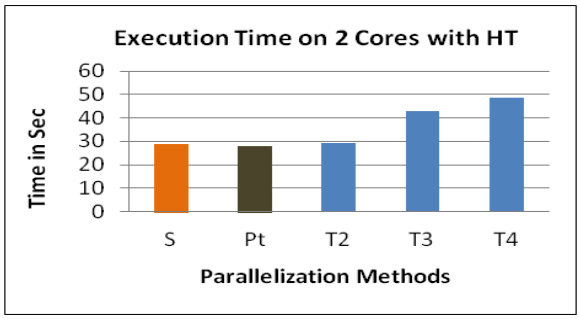
IV. RESULTS AND PERFORMANCE ANALYSIS

We have parallelized various programs using S2P tool to analyse performance benefits on multicore hardware. Out of test programs, we chose mp3 decoder program since it contains opportunity for task parallelization along with loop parallelization. Mp3 decoder is a utility for converting 'mp3' files into 'wav' files. Time required to decode mp3 file depends on size of input mp3 file. As size of input file increases, time required to decode the file also increases.

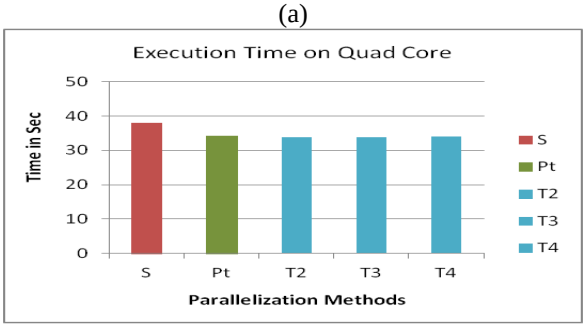
S2P generates parallelized code using Pthreads or OpenMP for task parallelization. We have taken these results on two different machines. One machine is an Intel Core i3, which has 2 cores with HyperThreading (HT) technology and frequency of 3.20 Hz. Other machine is an Intel Core 2 Quad, a four-core machine with frequency of 2.66 Hz. Operating system on both machines is Ubuntu 10.04. RAM of dual core machine is 3.1 GB and that of quad core machine is 2.9 GB. Dual core machine with HT contains two physical cores. However, a scheduler treats them as four logical cores. In case of logical processors in HT-enabled machine, the architectural state of the processor is duplicated. The architectural state consists of processor data registers, segment registers, control registers, debug registers, and most of the model specific registers (MSRs). However, quad core machine contains four independent cores, each having its own execution unit, cache, architecture state. We observed remarkable differences in the performance of parallelized code on these two machines.

In all of the following figures, 'S' column depicts the time required to execute the sequential code. 'Pt' column depicts the time required to execute the parallelized code, in which Pthreads are used. 'T2', 'T3' and 'T4' columns represent the time required to execute the parallelized code where the number of OpenMP threads used is 2, 3 and 4 respectively.

In case of parallelized code using pthreads, the number of threads created is equal to the number of tasks formed by S2P tool. There is no thread pooling and scheduling is managed by the operating system scheduler. In case of 'T2', 'T3' and 'T4', OpenMP 'task' constructs are used. In these cases, thread pooling is present and OpenMP manages it. Figure 5 shows the performance of mp3 decoder for an input file of size 6 MB.



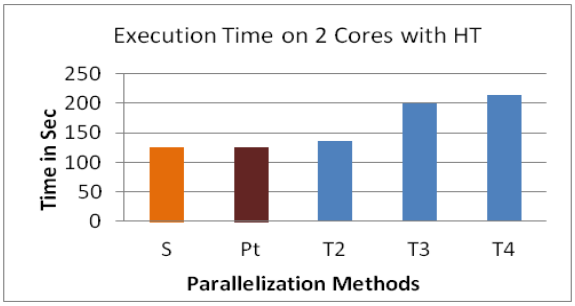
(a)



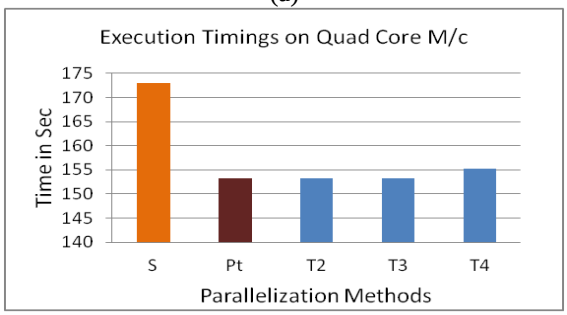
(b)

Fig.5. Execution timings of mp3 decoder on a file of size 6MB

As shown in figure 5 (a), time required to execute sequential code and parallel code using OpenMP tasks with 2 threads (T2) is approximately equivalent. Due to two threads, it executes concurrently and CPU utilization is more than 150%. In case of 'T2' and 'T3', the execution timings are further increasing. The possible reasons of not getting any performance gain could be more number of context switches and inconsistencies in data caching in HT environment. However, if we see similar execution on Quad core machine as shown in figure 5 (b), it is observed that OpenMP task achieved better results than serial execution. Due to more number of available cores, overhead of context switching is less. On both machines, it is observed that, parallelization with Pthreads shows performance benefit. However, the benefit is not significant, as thread pooling is not used. Figure 6 shows execution timings required to decode a file of size 27.7 MB on dual core with HT machine and on quad core respectively.



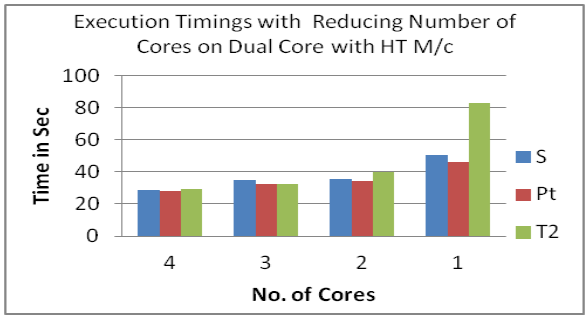
(a)



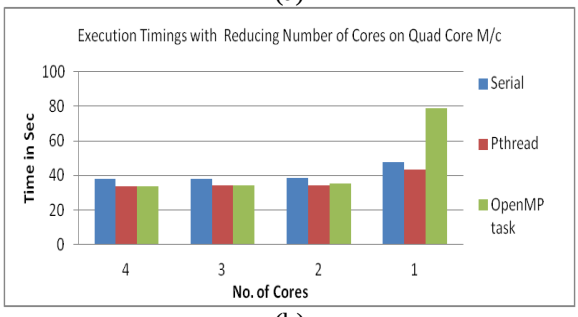
(b)

Fig.6. Execution timings of mp3 decoder on a file of size 27.7 MB

Figure 7 shows the execution timings of mp3 decoder on a file of size 6 MB on varying number of cores. We observed that the performance degrades as number of cores goes on decreasing. The performance degrades due to reduced computational power and increased context switching.



(a)



(b)

Fig.7. Execution timings of mp3 decoder with reducing number of cores

V. CONCLUSION AND FUTURE SCOPE

In order to increase the performance of software programs, parallelization is one of the important techniques. In contrast to the extensive efforts required for manual parallelization of programs, automatic parallelization tools are need of the hour. The S2P tool presented in this paper is a completely automated parallelization tool, which converts the sequential C programs into functionally equivalent parallel programs. To increase the applicability to larger set of codes, S2P tool presents task as well as loop level parallelization, as opposed to other available tools. Our observations on various test codes highlight the fact that the performance gain in parallelized code depends on the inherent parallelization degree present in the original sequential program.

The results presented in the previous section portray small part of performance experiments. Performance analysis of the S2P tool is still in progress. The key factor that contributes to the performance gain is minimizing the overhead of thread management during execution of parallelized code. Few relevant experiments in this direction include creation of thread pool for Pthreads and changing the granularity of tasks.

ACKNOWLEDGMENT

We would like to thank all the S2P project members, past and present, for their efforts. We would also like to thank the CREST team for their constant support and encouragement.

REFERENCES

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, "An Overview of the SUIF Compiler for Scalable Parallel Machines", Seventh SIAM Conference on Parallel Processing for Scientific Computing, 1995.

[2] http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/, accessed Oct 2011

[3] http://www.cri.ensmp.fr/pips/, accessed Oct 2011

[4] http://www.nvidia.com/, accessed Oct 2011

[5] A. Sane, P. Ranadive, S. Sah, "Data dependency analysis using data-write detection techniques", ICSTE 2010, Vol 1, V1-9 – V1-12.

[6] U. Banerjee, "Dependence Analysis for Supercomputing", Kluwer Academic Publishers, Norwell, MA, 1988.

[7] Deutsch, Alain, "Interprocedural May-Alias Analysis for pointer beyond k-Limiting", Proc. Of SIGPLAN '94, Conference of Programming language design and implementation, Vol. 29, No. 6, 1994, pp. 230-302.

[8] K. D. Cooper and K. Kennedy, "Fast Inter-procedural side-effect analysis in linear time", PLDI 1988, Vol. 23, Issue 7.

[9] V. Vaidya, S. Sah, P. Ranadive, "Optimal Task Scheduler For Multicore Procesor", ICSTE 2010, Vol 1, pp. V1-1 – V1-4.

[10] C. D. Offner, "Modern Dependency Testing", September 24, 1996, pp. 1-63.

[11] V.G. Vaidya, P. Agrawal, A. Athavale, A. Sane, S. Sudhakar, P. Ranadive, "Increasing Parallelism on multicore processors using Induced Parallelism", ICSTE 2010, Vol. 1, pp. V1-5 – V1-8.

ABOUT THE AUTHORS

Aditi Athavale

Aditi Athavale received her M. Tech degree in Information Technology from Indian Institute of Technology, Roorkee, India in 2009. She is currently working as a Junior Scientist at Center of Research in Engineering Sciences and Technology (CREST) at KPIT Cummins Infosystems Ltd., Pune, India. Her research interests include Cryptography and its applications, Multicore Programming, and Data Security.

Priti Ranadive

Priti has a M.Sc. from Pune University in the faculty of Instrumentation Science. She is currently working at a Scientist at Center of Research in Engineering Sciences and Technology (CREST) at KPIT Cummins Infosystems Ltd., Pune, India. She is also a research scholar at Symbiosis Innovation and Research Institute. Her areas of interest include Embedded Systems, OS and RTOS, Multicore programming and TRIZ.

M Niswanth Babu

He is an alumnus of IIIT-Hyderabad, India. He is working in KPIT Cummins as a Senior Research Associate for last 1.5 years. His areas of interest include Parallel Computing, Information Retrieval & Extraction.

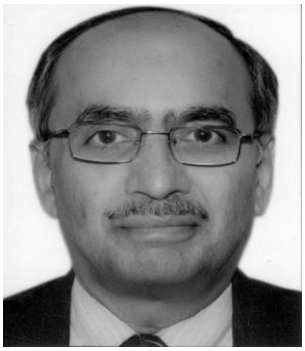
Prasad Pawar

Prasad Pawar received the M.E. degree in computer science from Walchand College of Engineering Sangali, Maharashtra in 2008. He is currently working as a Sr. Research Associate at KPIT Cummins Infosystems Ltd. Pune, India. His research interests include parallel computing, multicore programming, business continuity and disaster management, IP storage area network, network security.

Sudhakar Sah

Sudhakar Sah received his M. Tech degree from Indian Institute of Technology (Delhi, India) in 2005.He is pursing PhD in computer science from Symbiosis Institute of Research and Innovation (Pune, India). He has worked in embedded systems group at TCS (Bangalore, India) till 2005. He is currently a Senior Scientist at KPIT Cummins Infosystems Ltd. (Pune, India). His research interests are parallel computing, GPGPU, Signal and Image processing and Mathematical algorithms.

Dr. Vinay Vaidya



Dr. Vinay Vaidya received PhD in Computer Vision from the University of Washington in 1992. In the past, he has worked at Boeing (Seattle), Fujitsu (India) and Siemens (India). At present, Dr. Vaidya is the Chief Technology Officer and VP at KPIT Cummins, where he heads the Center for Research in Engineering Sciences and Technology (CREST). He is an honorary Professor of Computer Studies at the Symbiosis International University, India. He has earned a position in the special edition of Who's Who in the World, 2008. He has 12 patents and several research papers to his credit. His research interest includes Pattern Recognition, Image Processing, High Performance Computing, Multicore technology, Data Compression, and security systems.

Chaitanya Rajguru

Chaitanya Rajguru is a Technical Fellow at KPIT Cummins. He received his B.Tech. degree in Electrical Engineering from IIT Bombay, and his M.S. degree in Electrical Engineering from Virginia Tech, USA. He has over 19 years' professional experience at Intel Corp. and at KPIT Cummins. He holds six patents on circuits and systems. His interests include computing hardware and algorithms, VLSI technology, and energy & power in systems.