Volume 3, Issue 10, October 2013 **ISSN: 2277 128X**

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Effective Load Balancing Using Data Driven Threads in GCC

Praveen Kumar Reddy.M
School of Computing Science
India

Adithya Chandra Varma.D
School of Computing Science
India

Prof. Varun Kumar. M
School of Information Technology
India

Abstract*— *Stream programming restricts parallel processing that can be performed by applying a series of kernel functions to elements in a stream by compilers. It achieves this by decomposing the program into tasks and identifying the flow of data between them. Existing system provides an extension to OpenMP 3.0 in GCC compiler for enabling stream processing. But the system fails to support proper load balancing. Proposed system tries to overcome this limitation by making use of data driven scheduling by means of data flow threads implemented in the compiler. The data flow thread model extension is prototyped for GCC 4.2.

Keywords*— *GCC, OpenMP, stream programming, data-flow thread.

I. INTRODUCTION

Single-threaded application's performance is expected to change with present generations of CPU's [1, 3]. In order to improve performance one must change the code structure which is a complex task for memory hierarchies and parallel hardware [2, 5, and 6]. Also, it becomes a difficult task for programmers to increase the performance as it involves target optimization. So to address this problem high level languages are proposed to express communication patterns which are independent of hardware. In recent times stream programming was introduced which guaranties functional determinism in parallel programming. Stream programming allows applications to achieve parallel processing easily. Stream programming is used in applications where application needs more computational entities such as Floating Point Unit(FPU) on a Graphics Processing Unit(GPU) without managing synchronization explicitly[12,13,18].

Stream programming model simplifies the hardware and software by limiting the parallel computations. This can be achieved by applying a series of kernel functions to each element in a stream (data). Stream programming also allows pipeline, data and task parallelism by producer and consumer relationship [11, 12, and 17]. This relationship allows the applications to take scheduling decision easily. The main advantage of the stream programming relies on its kernel functions by which it achieves a better allocation of resources, better scheduling of global Input and output (I/O) communications. This feature of stream programming has been added to OpenMP 3.0 [4, 8, 9] for which a compiler should automatically support the stream programming qualities. OpenMP allows explicit data flow by using sharing clauses (shared and private data).To achieve this Manual synchronization (using shared memory) is necessary to use the task constructs, which is a complex task for developers [19, 17, 21].

This paper provides a proper load balancing support and a run time dataflow by making use of data driven scheduling by means of data flow threads .We implemented this paper and evaluated the results in GCC 4.2. This paper is planned as follows. Section 2 gives a detailed literature survey and the motivation to do this work. Section 3 and 4 presents our approach towards the proposed system and its advantages .Section 5 implements our proposed work in GCC 4.2 followed with results in section 6.Section 6 ends this paper with conclusion and future work.

II. EXISTING SYSTEM &RELATED WORK

Execution of parallel data leads to a need of high memory bandwidth for multi core architectures, hence leading to access latencies in main memory [2, 3, and 4]. To address this problem stream programming has been introduced. In stream programming data and task level parellism is achieved by its high level semantics. Many libraries and languages are developed for the stream programming [5, 6, and 8]. Some languages are common programming languages that are independent of architecture while others are specially designed for particular architectures [7]. A stream programming is done by dividing the program into pipeline stages. These stages can be either sequential or parallel depending upon the producer and consumer correlation. In [17] author proved by his results that application can be parallelized by using pipeline parallelism by which he achieved scalability. This pipeline parallelism is introduced in the multicore environment to increase the efficiency and speedup by dividing the application into parallel tasks. But in [17], including pipeline parellism in OpenMP does not provide good results forparellization of Bzip2 due to the data dependency between tasks. Instead they used FIFO queues to parellize the tasks. In this paper data dependency between the tasks is achieved by means of providing data flow threads.

Stream It language [12] is a parallel programming language that is specially developed for huge data (streaming) applications. The main advantage of using this language is providing explicit parellism, independent of the architecture [13]. Stream it allows explicit parellism by Split Join concept, by dividing the output of the application in to different streams, then joining the results in to a single output stream [11].

In [4] author proposed Brook language with SPMD (Single Program Multiple Data) operation on the stream. This language is an extension added to C which ensures parellism using control flow operations. Their system design was assumed to be light weight execution model that provides a proper load balancing but limited for small applications [11, 13]. The existing system provided an extension to OpenMP to ensure stream programming [25, 26]. This extension has been done because of the increase in parallel applications and also to address the scalability and efficiency issue in the modern multi-core architectures [21, 22, 23, and 24]. The existing system also offered the principles needed to extend this stream programming in OpenMP and its advantages. Proper static scheduling and task free implementations were proposed to ensure persistency among the tasks. In the need for ensuring persistency among the tasks, the system failed to address proper load balancing.

III. PROPOSED SYSTEM

The proposed system uses the properties of SSA (Static single Assignment) to achieve thread level data flow parellism. Static Single Assignment (SSA) is the property of IR where each variable should be assigned exactly one time.
The proposed system used four built in functions in the GCC compiler as a pass to manage the frames and threads.

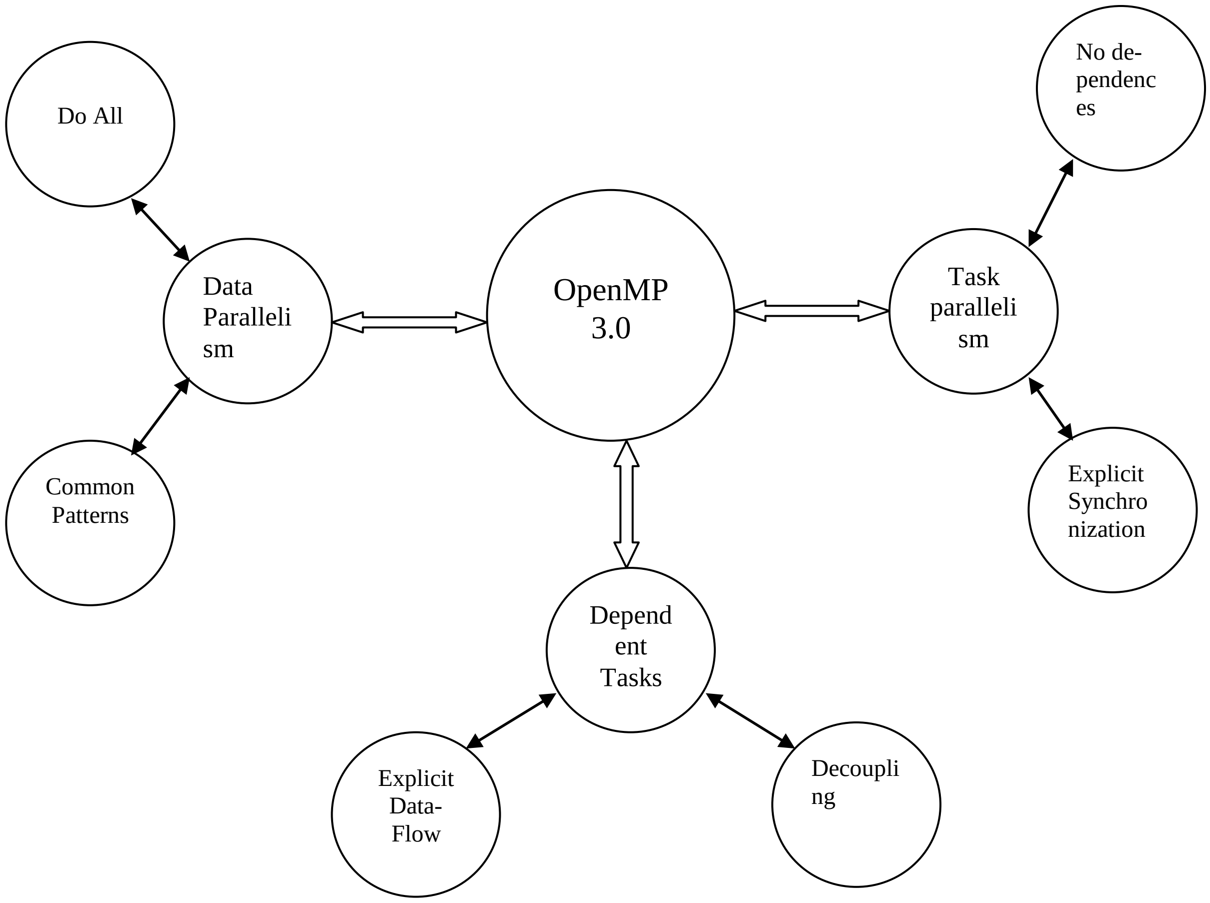

Figure 1: Architecture of the Proposed System

Table 1: Functions to manage Frames and Data Flow

Functions	**Description**
void *tacreate(void (*fun)(), int sc, int size);	A new data flow thread is created and its associated frame is assigned to the thread
void tdecrease(void *fp);	After the termination of the thread the frame pointer is decremented by one.
void ternd();	The frame is deallocated for the thread that is currently allocated
void *taget cfp();	Current Thread's frame pointer is returned.

IV. DATA FLOW AT THREAD LEVEL

Conversion of sequential programs in to parallel data flow involves splitting the basic blocks at statement level there by reducing the communication overhead. In order to reduce the communication overhead, SCC (Strongly Connected Components) should be identified using dependency graph. The proposed algorithm uses SSA form and generates a pass to the GCC.

*A.*DATA FLOW THREADS GENERATION ALGORITHM

Data Flow Threads Pseudo code

```
Input: Application program
Output: Program Dependency graph, Data Flow Program Dependency Graph
Data Flow Thread Generation (Application program)
Begin
        If (serial program) then
                Generate PDG (Program Dependency Graph) using Static Single Assignment.
                Apply the functions and merge the Strongly Connected Components (SCC) in the PDG
                Define DFPDG (Data Flow Program Dependency Graph), which is derived from (SSAPDG) to assign
                the data flow frames and flow of values with respect to control dependencies
                Generate the code for target data flow using Data Flow Program Dependency Graph
        End of if
End
End of Data Flow Thread Generation
```

Fig 1.1 Pseudo code of the Data Flow Thread Generation

B. APPLYING SSA TO THE BASIC BLOCKS

A unique name representation is given to each variable in the figure 1.When multiple inputs come to a single basic block node is introduced which eliminates redundancy. Some of the Ω nodes may lead to redundancy, for which Loop un-switching is done. The Ω node defined x, y is used only outside of the loop, whereas the Ω node for variable j is used inside the while loop leading to redundancy. Hence loop un-switching is done.

```
T1      x=0;                            T1      x=0;
T2      j=0;                            T2      j=0;
T3      y=0;                            T3      y=0;
T9      while (j<100)                   T9      if (j<100) {
        {                                       Do {
T6              x=j;                    T6      x=j;
T7              y=fun (j);              T7      y=fun (j);
T8              j=last (j);             T8      j=last (j);
        }                               T9      } while (j<100)
T12     if(x>y)                                 }
T13     set=x;                          T12     if(x>y)
        Else                            T13     set=x;
T14     set=y;                          T14     else set=y;
T16     return set;                     T16     return set;
```

Figure 2. Before Unswitching after Unswitching

```
T1      x0=0;
T2      j0=0;                           T10     x2=Ω(x0, x1);
T3      y0=0;                           T11     y2=Ω (y0, y1);
T4      if (j0<=99)                     T12     if(x2>y2)
        goto T5;                                goto T13;
        Else                                    else
        Goto T10;                               goto T14;
T5      j1=Ω (j0, j2);                  T13     set0=y2;
T6      x1=j1;                                  goto T15;
T7      y1=fun (j1);                    T14     set1=y2;
T8      j2=last (j1);                   T15     set2=Ω (set0, set1);
T9      if (j2<=99)                     T16     return set;
        Goto T5;
        Else
        Goto T10;
```

Figure 3.SSA after Unswitching

C. GENERATION OF PROGRAM DEPENDENCY GRAPH FOR SSA

Generating program dependency graph for the figure 3 is mainly for two reasons

a) To remove the redundancy between the variables and maintain each definition unique.

b) The complexity is reduced from O (n^2) to O (n), since the redundant edges in the PDG can be eliminated.

Figure 3 represents the program dependency graph (PDG) for the SSA code of the figure 2, where each statement is represented in the square box and its dependencies are represented in the form of edges. Solid lines in the figure denote the data dependencies between the statements whereas dotted line represents control dependencies.

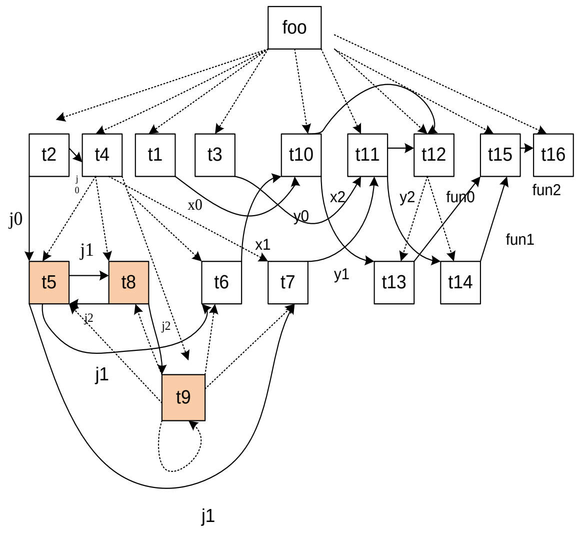

Figure 4: Program dependency graph (PDG) for the SSA code of figure 3

D. Merging SSC's (Strongly Connected Components)

Since the statements t5, t8, and t9 are strongly connected components (they cannot be parellized and their execution is serial) replace with a new component scc1 which reduces the overhead for parellization and thereby providing a proper load balancing. Figure 4 represents the PDG for the SSA after merging strongly connected components.

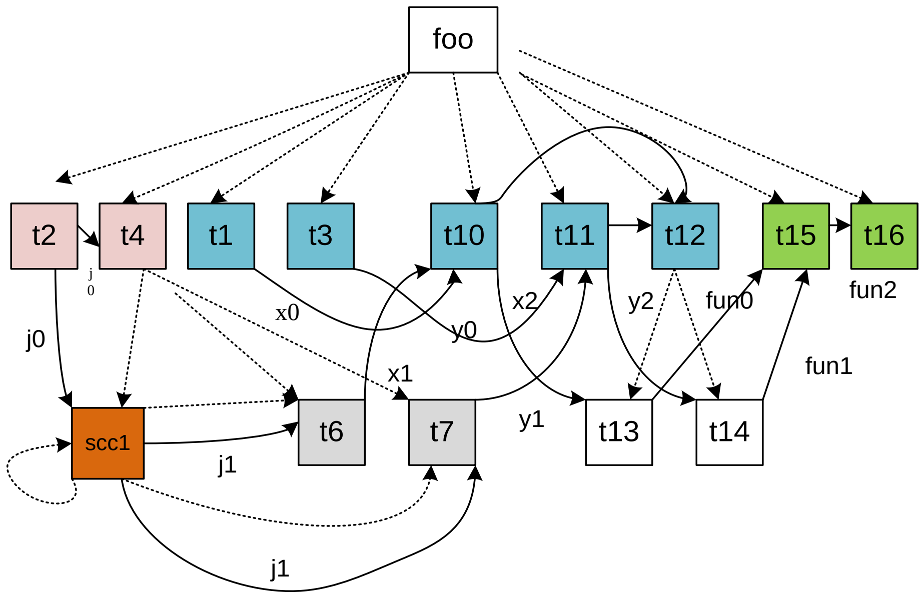

Figure 5: Generation of PDG for SSA after merging Strongly Connected Components (SCC)'

E. GENERATION OF DATA FLOW PROGRAM DEPENDENCE GRAPH

From figure 5 the nodes t6 and t7 are of same type since their control depends on t4 and hence t6, t7 can be fused to a single node. Similarly we can observe that t2, t4, t1, t3, t10, t11, t12, t15, t16 can be fused to single node. But this is a critical thing as there is chain t10, t13; t15 where t13 depends on t10 and t15 depends on t13.Similar case is applicable to the chain t11, t14, t15. Hence typed fusion is applied to such chains. Figure 6 represents the data flow program dependence graph after applying typed fusion with a restriction that dependence chains are not merged as a single component.

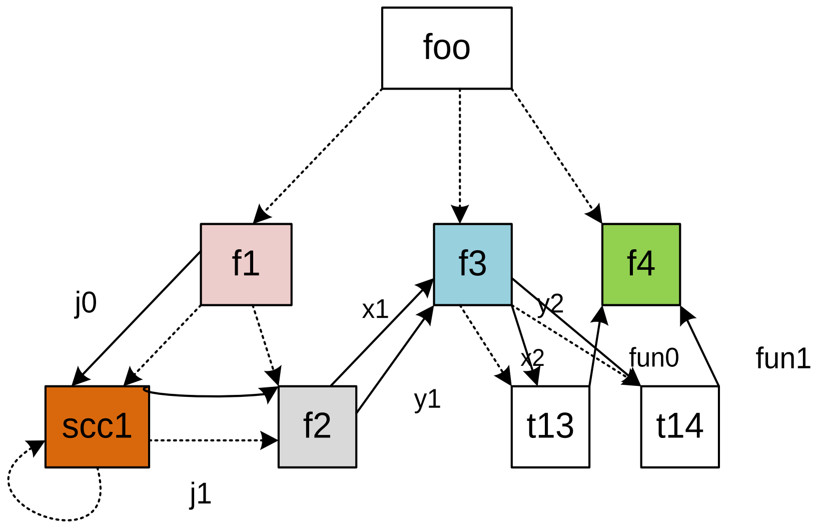

Figure 6: Generation of Data Flow Program Dependence Graph for Figure 5

V. EXPERIMENTAL ANALYSES

The experimental analysis describes the implementation of thread level data flow approach and its comparison with serial data flow. This thread level data flow approach is implemented in GCC. The libgomp pass of GCC was modified to incorporate the necessary changes to support data flow threads and frames.

Table 2: Comparison of execution time of with and without thread level data flow in dual core processor

Block size	Execution Time without data flow at thread level(in Seconds)	Execution Time using data flow at threads level(in seconds)
b7	400.324518	382.87848
b8	400.324518	363.685213
b9	400.324518	377.539737
b10	400.324518	391.3115

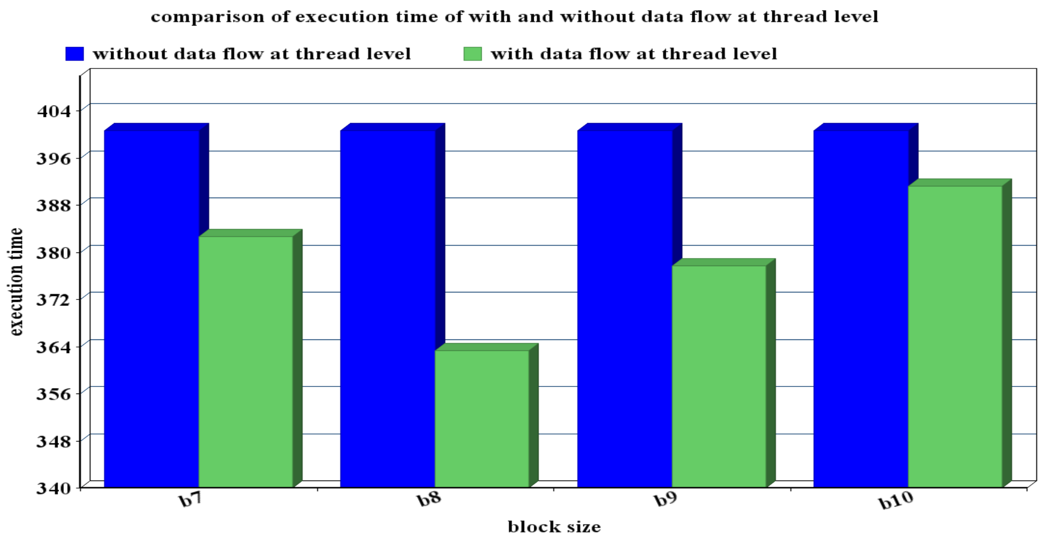

Figure 7: Comparison of execution time for sequential approach with Thread level Data flow approach for dual core processor.

Table 3: Comparison of execution time of with and without thread level data flow in quad core processor

Block size	Sequential Execution Time(in Seconds)	execution time using data flow at threads level (in seconds)
b7	387.728347	352.973301
b8	387.728347	340.485913

b9	387.728347	356.321737
b10	387.728347	360.312115

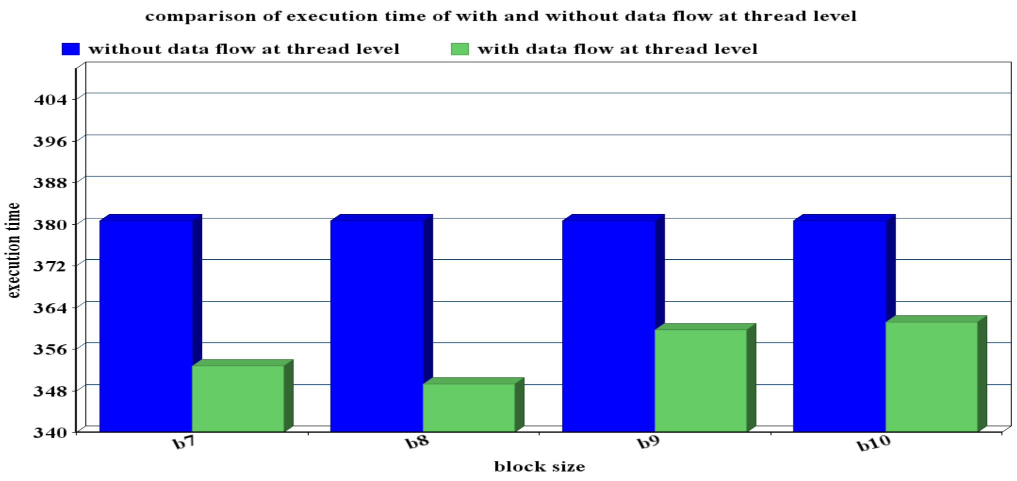

Figure 8: Comparison of execution time for sequential approach with Thread level Data flow approach for quad core processor.

The analysis was conducted with the Cholesky algorithm with a square matrix of 2^{12} x 2^{12}size with 20 iterations. The results were obtained by making use of varying block sizes (from 2^7 - 2^{10}).

Block size	Speed up without using data flow threads(in Seconds)	Speed up using data flow at threads level(in seconds)
b7	2.523	2.689
b8	2.523	2.931
b9	2.523	2.834
b10	2.523	2.634

Table 4: Comparison of speed up of with and without thread level data flow

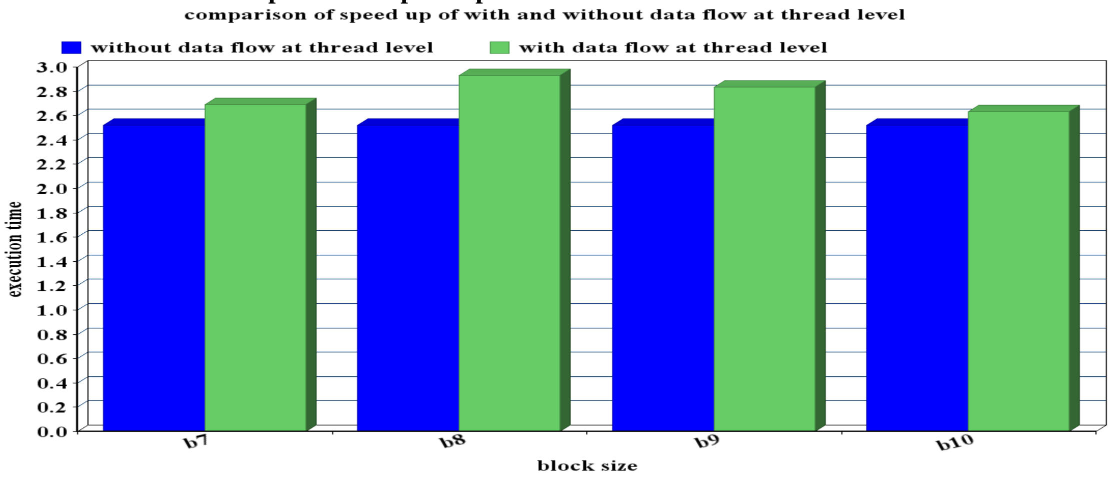

Figure 9: Comparison of speed up of with and without thread level data flow

VI. CONCLUSIONS

This paper provides a proper load balancing in stream programming by making use of data driven scheduling which is implemented in the GCC compiler. The proposed algorithm initially generates a PDG using SSA (Static Single Assignment) then applying the loop unrolling. The DFPDG is generated using typed fusion. The experimental results show that the proposed algorithm improves the efficiency and decreases the execution time, thereby providing a proper load balancing.

ACKNOWLEDGMENT

The authors would like to thank the School of Computer Science and Engineering, VIT University, for giving them the opportunity to carry out this project and also for providing them with the requisite resources and infrastructure for carrying out the research.

REFERENCES

[1] Arvind, R. S. Nikhil, and K. Pingali. I-structures: "Data structures for parallel computing". ACM Trans. *On Programming Languages and Systems*, 11(4):598–632, 1989.

[2] Dr.M.RajasekharaBabu, Tutorial/Lecture Notes on "*Advanced Computer Architectures and OpenMP*" Fall Semester 2012", VIT University, India.http://mrajababu.com.

[3] D r.M.RajasekharaBabu Babu, Tutorial/Lecture Notes on "*Advanced Compiler and Design*" winter Semester 2012", VIT University, and India.http://mrajababu.com

[4] Antoniu Pop and Albert Cohen. *"*OpenStream: Expressiveness and data-flow compilation ofOpenMP streaming programs*"*. ACM *Transactions on Architecture and Code Optimization (TACO),* selected for presentation at the HiPEAC 013 Conf., January 2013

[5] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta.CellSs: "a programming model for the Cell BE architecture". In Super Computing,pages 94–110,2006.

[6] R. H. Bisseling. *Parallel Scientific Computation: "*AStructured Approach using BSP and MPI". Oxford University Press, pages 41-48, Mar. 2004.

[7] P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, andE.Ayguadé.*A streaming machine description andprogramming model*. In Soft Computing, pages 107–116, 2007.

[8] R. Cytron. Doacross*: Beyond vectorizationformultiprocessors*. In Intl. In preceding ofConf. on Parallel Processing(ICPP),pages 33-46, Saint Charles, IL, 1986.

[9] J. B. Dennis and G. R. Gao. *Efficient pipelined dataflowprocessor architecture*. In Supercomputing (SC'08), pages 368–373, 2008.

[10] H. M. et al. ACOTES project: *Advanced compilertechnologies for embedded streaming*. Intl. J. of ParallelProgramming, 2010.Special issue on European HiPEACnetwork of excellence member's projects.

[11] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero.*Effective communication and computation overlap withhybrid* MPI/SMPSs. In Parallel Computing, 2010.

[12] A.Munshi. *The OpenCL specification*, v. 1.0, revision29.www.khronos.org/registry/cl/specsopencl-1.0.29.pdf, 2008.

[13] V. Pankratius, A. Jannesari, and W. F. Tichy.*Parallelizingbzip2: "*A case study in multicore software engineering". IEEE Software., page no 70–77, 2009.

[14] A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. J. Kelly.`"Improving GNU compiler collection infrastructure forstreamization". In Proceedings of the 2008 GCC Developers'Summit, pages 77–86, 2008. http://www.gccsummit.org/2008.

[15] The StreamIt language. http://www.cag.lcs.mit.edu/streamit/.

[16] E. A. Lee and D. G. Messerschmitt. "Static scheduling of synchronous data flow programs for digital signal processing". *IEEE Trans. Computers*, 36(1):24–25, 1987.

[17] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. *Automatic Thread Extraction with Decoupled Software Pipelining*.In IEEE/ACM Intl. Symp.on Microarchitecture, volume 0, pages 105–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[18] J. Planas, R. M. Badia, E. Ayguad´e, and J. Labarta.*"*Hierarchical Task-Based Programming With StarS*s"*. Intl. J. on High Performance Computing Architecture, 23(3):284–299, 2009.

19] A. Portero, Z. Yu, and R. Giorgi. T-Star (T*): "An x86-64 ISA extension to support thread execution on many cores". *In HiPEAC ACACES- 2011*, pages 277–280, Fiuggi, Italy, July 2011.

[20] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. *"*Parallel-stage decoupled software pipelining". In Proc. of *the 6th annual IEEE/ACM Intl. Symp.on Code Generation and Optimization, CGO '0*8, pages 114–123, New York, NY, USA, 2008. ACM.

[21] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. TFlux: "A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems". In Intl. *Conf. on Parallel Processing (ICPP'08*), pages 25–34, Portland, Oregon, Sept. 2008.

[22] P. Tu and D. Padua."Gated SSA-based demand-driven symbolic analysis for parallelizing compilers". In Proc. of the 9th Intl. Conf. on Supercomputing, ICS '95, pages 414–423, New York, NY, USA, 1995. ACM.

[23] R. Cytron, J. Ferrante, and V. Sarkar. "Experiences using control dependence in PTRAN*". In Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 186–212, London, UK, UK, 1990. Pitman Publishing.

[24] A. L. Davis. The architecture and system method of DDM1: "*A* recursively structured data driven machine". In Proceedings of the *5th annual symposium on Computer architecture,* ISCA '78, pages 210–215, New York, NY, USA, 1978. ACM.

[25] Roberto Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", *ACM Computing Frontiers*, Cagliari, Italy, May 2012, pp. 303-304.

[26] Roberto Giorgi, Zdravko Popovic, Nikola Puzovic, "*DTA-C:* A Decoupled multi-ThreadedArchitecture for CMP Systems", *Proc. IEEE SBAC-PAD*, Gramado, Brasil, Oct. 2007, pp. 263-270