Compiler Assisted Parallelization and Optimization for Multicore Architecture

THESIS

Submitted in partial fulfillment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

D.C.KIRAN

Under the Supervision of

Prof. S. GURUNARAYANAN

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

2014

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)

CERTIFICATE

This is to certify that the thesis entitled **“Compiler Assisted Parallelization and Optimization for Multicore Architecture"** submitted by **D. C. Kiran** ID. No. **2007PHXF013P** for award of Ph.D. Degree of the Institute, embodies original work done by him under my supervision.

(Signature of the supervisor)
Prof. S Gurunarayanan
Professor
Department of Electrical Electronics and Instrumentation Engineering
Dean Work Integrated Learning Programmes Division
Birla Institute of Technology and Science-Pilani
Pilani – 333 031 (Rajasthan) INDIA

Date:

Dedication

To my father Channaiah and mother Sulochana who believed and allowed me to achieve whatever I thought in my life.

Acknowledgements

Thanking my god who kept his promise by being with me throughout my research expedition, I would like to acknowledge the efforts of my supervisor Prof. S. Gurunarayanan who has been more to me than just an adviser in technical matters. A friend, a mentor, and a guide for life, he gave me the flexibility to pursue whatever I felt was appropriate, provided me with continuous guidance even beyond his areas of interest to help me work efficiently and remain focused, and gave me the ability to form a vision.

I am very thankful to Prof Janardan Prasad Misra who stepped in at the right time to work with me, and without his continuous encouragement, inspiring dedication, and organized approach to research, I couldn't have completed this thesis. He is the most sincere teacher and guide I have come across, and I consider myself extremely lucky to have gotten his attention.

Special gratitude to Prof Sudeept Mohan, the ACO course for which I was attached with him gave me a deep understanding of processor design. He has also been kind to provide useful suggestion and feedback on all part of my work.

I would like to thank Prof. B. N. Jain, Vice Chancellor, Prof. G. Raghurama, Director, Prof. S K Verma, Dean, Academic Research Division, Prof Rahul Banerjee, Head of Department, Computer Science & Information Systems, Birla Institute of Technology and Science, Pilani (BITS-Pilani), (Raj.), for giving me an opportunity for PhD program.

Further, I sincerely acknowledge the encouragement and help received from Prof Shanmuga Sundar Balasubramaniam and Prof. Navneet Goyal, at various stages of the work.

My wife N. Mehala with whom I closely shared much of my experience, especially as PhD student, the contradictions, the disappointments at innumerable paper rejections, the sporadic joyous moments at papers acceptances, probably she is the only one who understands what made this road really long, and gave me company all along.

To my son Mano Srijan who missed many stories when I worked till late night. Over the last almost-nine enriching years at BITS Pilani, so many other people and things have been a part of this experience as well, that it is hard to choose and name only a few. My colleagues in department and friends and the time we spent together will forever remain precious to me.

Abstract

Compiler Assisted Parallelization and Optimization for Multicore Architecture

Continuous improvement of VLSI technology coupled with need for faster processing capability has led to several innovations in the field of computer architecture resulting in development of multicore processors. A multicore processor has multiple processor cores on a single chip. Each individual core has separate register file and is capable of executing complete ISA (Instruction Set Architecture). In order to exploit the computing capabilities of multicore processors, significant amount of research in the area of code parallelization and multiprocessing has been carried out. An application running on a multicore system does not guarantee the performance improvement until the application has been explicitly designed to take advantage of multicore processor. To develop an application that exploits computing capabilities of multicore, two approaches are followed. The first approach is to develop an explicitly parallel code that can be scheduled on multiple cores of a given processor and the other approach is using a compiler to extract fine grained parallelism by identifying the sets of instructions that can be executed in parallel. Current focus by researcher and programming language developers is to exploit coarse grain thread and data-level parallelism. There is very little effort from the research community toward the exploitation of compiler driven fine grained parallelism of a sequential program.

The multicore processors can be made to exploit fine grained parallelism of a given code by exposing the low level architectural details to the compiler and operating systems. Several multicore architectures are proposed and are being designed such that it supports the minimal set of operations required for executing an instruction, and other tasks including extracting the fine grained parallelism are left for compilers and run time environment. The runtime environment can manage resource allocation, extracting parallel constructs for different cores, and scheduling based on information generated by the compiler.

The challenge in achieving a performance gain from fine-grain parallelism is identification of the fine grained thread from a given single threaded application and scheduling these threads on different cores of the multicore processor.

To avoid the congestion on small shared register file as in other parallel architectures, the memory hierarchy of multicore architecture generally has private register files. The fine grained threads that are scheduled on to different cores need to be allocated registers from respective register file of the core on which they are scheduled.

To effectively utilize the potential benefits of the multi-core processor, the thesis focuses on improving performance through automated fine-grain parallelization, where a sequential program is split into parallel fine grained threads and are scheduled on to multiple cores. It is also proposed to develop register allocation strategy for fine grained threads which are scheduled on multicore processor. The register allocation is performed by considering individual register files of each core of multicore processor.

This thesis modifies the flow of current compiler by splitting the sequential program to create fine grained threads, proposes five scheduling heuristics (1 local and 4 global), and register allocating heuristics for fine grained threads which are scheduled on multiple cores. The work is evaluated using speed-up, power consumption, performance per power, communication cost, and spilling as metrics. The RAW benchmark suite is used to compare the results.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

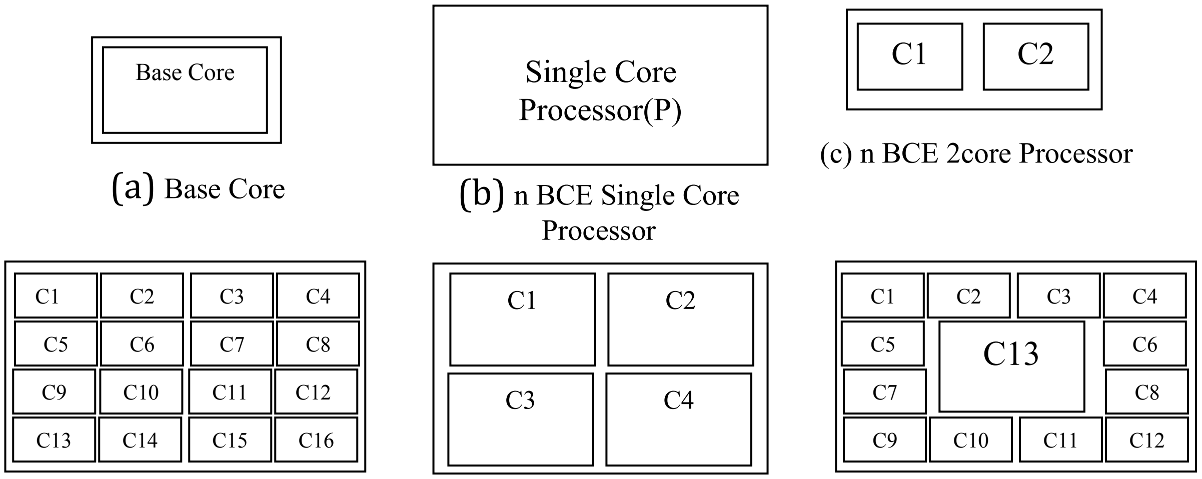
Abbreviation	Details or Expanded Form
ISA	Instruction Set Architecture
BCE	Base Core Equivalent
ILP	Instruction Level Parallelism
DAG	Directed Acyclic Graph
CFG	Control Glow Graph
QUAD	Three address code format of instruction
ARENA	Name of structure of Basic Block of CFG
SSA	Static Single Assignment
SAME	A function used to identify copy related instructions
Phi (ϕ)	A data structure which stores the information of copy related instructions.
DSB	Dependent Sub-block First Scheduler
MDS	Most Dependent Sub-block First Scheduler
LLSF	Least Latency Sub-block First Scheduler
HIB	Height Based Scheduler
IBS	Intra Block Scheduler
IRS	Integrated Register Allocation and Scheduling
SB_i	ith Sub-block
B_p	pth Basic block
SB_iB_p	ith Sub-block in pth Basic block
$Perf_m(r)$, $Perf_s(r)$, $Perf_d(r)$, $Perf_q(r)$	Performance on multicore, single core, dual core, and quad core processor respectively.
$Pw_m(r)$, $Pw_s(r)$, $Pw_d(r)$, $Pw_q(r)$	Power consumed by multicore, single core, dual core, and quad core processor respectively.
P_{fi}	Power consumed by cores of powerful multicore processor in idle state
P_{li}	Power consumed by cores of less efficient multicore processor in idle state
P_d	Extra power consumed by dual core processor to execute non-parallelized program
P_q	Extra power consumed by quad core processor to execute non-parallelized program
w_c	Total power consumed by a core in multicore processor to execute non-parallelized program

Abbreviation	**Details or Expanded Form**
TRdy	Ready Time
TFns	Finish Time
TSct	Schedule Time of Core
TSch	Schedule Time of Sub-block
$Height_i$	Height of ith sub-block
L_i	Latency ith sub-block
C_i	Total Cycle time required by ith sub-block
TIc	Total Instruction Count
$Rreq_i$	Register Required by ith sub-block
HSB_jB_P	jth Hyper sub-block of pth basic block
$Ravl_j$	Register available in jth hyper sub-block

CHAPTER 01 *Introduction*

Coupled with technological advancement in the field of computer architecture and relentless demand for faster processing has led to development of several innovative technologies and products. Semiconductor industry had kept pace with Moore's law in terms of doubling the number of transistor on a chip and increased clock speed [1]. R. Dennard, et al., in 1974 proposed that with scaling ratio of 1/√2, the transistors count will double on a chip and clock frequency can be increased by 40% keeping the power consumption constant [2][3]. But with current feature size, the Dennard's law does not hold true any longer [4]. The continuous increase in number of transistor and clock speed has thrown design challenges for handling larger amount of heat dissipation. The power dissipation and thermal issues severely restricts the ability to continuously increase operating clock frequency of a processor [5][6]. This has led to development of homogeneous multicore architecture. A multicore chip supports multiple processor core on a single chip. The idea was to replace power hungry powerful processor with less powerful multiple cores [7][8]. Such developments lead to greater focus on exploiting the explicit parallelism by executing multi threaded applications or multiple tasks on multiple processor core to gain performance [9]. Since the approach tries to exploit explicit parallelism, the processor cores can be operated at lower clock frequency to achieve the same or better performance as compared to single core processor operating at higher frequency thus solving the heat dissipation problem.

The cores on a chip can be homogeneous or heterogeneous [10]. In case of homogeneous multicore processor chip, each core is equally capable and therefore allows any thread to execute on any core. Figure 1 provides an overview of the most common design of on chip multiprocessors used in today's system. Figure 1.c, Figure 1.d and Figure 1.e, depict homogeneous multicore processors and Figure 1.f is an example of heterogeneous multicore processor. A homogeneous architecture is undoubtedly easier to program for parallelism, because a program can make use of the all cores than in a heterogeneous architecture where all the cores do not support the same instruction set.


Base Core
Single Core Processor(P)
C1 C2
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16
C1 C2 C3 C4
C1 C2 C3 C4 C5 C13 C6 C7 C8 C9 C10 C11 C12

(a) Base Core

(b) n BCE Single Core Processor

(c) n BCE 2core Processor

(d) n BCE 16 core Homogeneous Multicore Processor

(e) n BCE 4 core Homogeneous Multicore Processor

(f) n BCE (12+1) core Heterogeneous Multicore Processor.

Figure 1. n Base Core Equivalent Processors

The design philosophy of multicore processor says that the cores on chip should be resource equivalent and power equivalent [11]. The Base Core in Figure 1.a, is a unit core made up of some r resources and consume some power to achieve some performance. A n BCE (base core equivalent) processor P is made up of r times the resources used for base core and consume k power budget to get the performance Perf(r). To build a n BCE processor r and k should be shared equally. If r of a single core is increased, sequential performance is increased. If r is distributed among multiple execution units, parallel performance is increased. So in multicore processor r resources are distributed to achieve n BCE processor. A homogeneous multicore processor can have n/r cores to have n BCE processor. For example, a single core processor with capability of 16 BCE (1*16 BCE) or a homogeneous multicore processor with 16 BCE cores (16*1 BCE) or a homogeneous multicore processor with 4 4BCE cores (4*4 BCE) are equivalent. Similarly, heterogeneous multicore processor 12*1 BCE + 1*4 BCE processor is equivalent to 16 BCE in terms of resources and power consumption.

The performance increases as number of cores increases, that is, $Perf_q(r) > Perf_d(r) > Perf_s(r)$. Where $Perf_s(r)$, $Perf_d(r)$ & $Perf_q(r)$ are performance of single core, dual core and quad core processors respectively. Ideally, $Perf_d(r)$ is $2*Perf_s(r)$ and $Perf_q(r)$ is $4*Perf_s(r)$. But according to Amdahl's law [12], performance (Pref) from N number of cores depends on a fraction f ($0 \le f \le 1$) of computation that can be parallelized. The fraction f is also responsible for increase in power consumption. The challenge of multicore programming involves making the fraction f equal or closer to 1. The challenges and issues associated with the multicore environment are discussed in next section.

1.1 Challenges In Multicore Environment

i. **Parallelizing the Sequential Program**

Though multicore technology offers clear benefits against the single core processor, the general understanding of researchers is that finding an effective way to exploit the parallelism or concurrency inherent in an application is one of the most daunting challenges. The multicore processors are general extension of shared memory multiprocessors whose computation power can be utilized effectively only by the applications with coarse grain threads. These designs provide real benefits for server-class applications that are explicitly multi-threaded. However, for desktop and other systems where single-thread applications dominate, multicore systems are yet to offer much benefit. There is a mismatch with current multicore hardware and applications, as most of the applications are single threaded and are unable to exploit the fine grained parallelism offered by multicore processors. To fully exploit the architectural capability and inherent fine grained parallelism of an application, it is desired to have parallel code. Writing parallel code is a tedious and requires expertise. Most of the features provided by explicit programming languages concentrate on parallelizing loops or iterative statements. It is essential to develop or convert existing sequential codes to parallel implementations. The support from compilers and run-time systems for the development of parallel application for multicore is vital.

ii. **Memory Management and Data Communication**

Memory hierarchy of multicore architecture generally has shared memory, second level shared cache, first level private cache, and private register files [13]. Issues related to memory can be classified as follows,

- **Register allocation**

  To avoid the congestion on small shared register file as in other parallel architecture (Pipelined, VLIW), the memory hierarchy of multicore architecture generally has private register files. The threads that are scheduled on to different cores need to be allocated registers from respective register file of the core on which they are scheduled.

- **Memory bandwidth**

  Memory bandwidth remains the bottleneck on multicore platform, although computing is cheap since there are many processing cores [14]. The existence of the memory bandwidth bottleneck is because of the use of shared bus by all CPU cores. Efficient memory management is very critical for a scalable application on multicore CPU.

- **Locality of reference**

  In a multicore processor with non-uniform cache architectures with distributed cache banks, data access latency is a limiting factor to performance. To mitigate this effect, it is necessary to leverage the data access locality and choose an optimum data placement so that the volume of inter-core messages is minimized. This requires a study of data accesses behaviors among multiple cores.

- **Memory Contention**

  Memory systems have been under a lot of pressure to keep up with the increasing demand for parallelism [15]. Memory Contention increases the need of synchronizing data among different cores, which has a big performance penalty because of bus traffic contention, locking cost and cache miss. Lock based synchronization has several limitations, including sensitivity to preemption and possibility of deadlock. A synchronization approach without lock is desirable. In multicore environment lock free synchronization is achieved using transactional memories. Regardless of which synchronization (lock based or non lock based synchronization) is used contention over shared data hamper the scalability.

  The current cache hierarchy has been unable to support high level demand for parallelism. Existing architectures employ lock-up free caches to avoid stalling the CPU and allow the cache miss to be serviced in the background. The *Miss Information/Status Holding Register* (MSHR) Files are responsible for keeping track of the outstanding concurrent misses. These types of caches are very costly in terms of chip area and power usage. This limits the size of the MSHR file that can be included, even for today’s large transistor budgets. For example, the L1 cache of an Intel Pentium 4 processor supports only eight outstanding misses.

iii. **Scalability**

Traditionally only super computers and high end servers needed major software scalability work, as they used many CPU sockets. The major scalability work was not needed in the low end computer systems as they had less CPU's. Scaling up the number of cores in multicore processors has provided a new dimension to scale up performance and requires extensive scalability work.

The factors which stop scalability are listed below.

- Programs may not inherently exhibit parallelism.
- Application program cannot scale up to meet the time bound constraints due to some physical constraints like memory. As number of cores increase, memory contention may increase leading to sequential access of data by deteriorating parallelism.

The expectation w.r.t increase in performance in multicore era is kept alive by the recent study on reevaluating Amdahl's law [16][17].

iv. **Power Consumption**

It was expected that the power consumption will remain same with the paradigm shift from single core to multicore processor, as the r resources used to design a n-BCE single core processor is distributed to design multicore processor with multiple slow cores (reduced clock speed). Though the slow cores of a multicore processor are energy efficient, the combined power consumption of cores is increased when used for parallel execution than sequential execution on single core processor to complete the task.

The fundamental reasons for increased power consumption are as follows.

- The core is less powerful (runs at reduced clock) than n BCE single core so it takes more time to execute thereby may consume more power.
- The second reason is due to Amdahl's law. According to Amdahl's law, the performance (Pref) from N of cores depend on fraction f ($0 \leq f \leq 1$) of computation that can be parallelized.

The power model proposed by Woo-Lee suggest [12] that if a program is executed on a single core processor the power consumption is $Pw_s = 1$. If the same program is executed on a multicore processor with n cores by parallelizing it 100% i.e., f=1 power consumption should be n*w_c where w_c is power consumed by slow core in multicore processor with n cores. But based on f and reduced strength (reduced clock) the power consumption of dual core processor should be ($Pw_s \le Pw_d \le 2\ Pw_s$). Similarly, when executed on quad core processor it is ($Pw_s \le Pw_d \le 4\ Pw_s$).

The power consumed by single core, dual core and quad core processor to execute the sequential and parallel version of the same program is summarized in Table1. Let P_{fi} & P_{li} be the power consumed in idle state by powerful core & less efficient core respectively. Let p_d & p_q be the extra power consumed by the less efficient n-BCE dual core and n-BCE quad core processor respectively and w_c be the total power consumed by each core of less efficient n-BCE multicore processor.

TABLE I. POWER CONSUMPTION COMPARISON

		Cores when fully utilized.	When sequential program runs on single core keeping other cores idle.	Ideal power consumption When sequential program (parallelized) runs on all the cores.	Time	Energy
Real	Single core	Pw_s = 1 W	Pw_s = 1 W	Pw_s = 1 W	5	5
Theoretical	Dual core as powerful as single core	2* Pw_s =2 W	1 + P_{fi} W	0.5+ 0.5 = 1 W	2.5	2.5
Expected	Dual core 50% less powerful than single core (Expected) i.e. w_c =0.5	2*(w_c) + 2 p_d = 1+ 2 p_d W	w_s + P_{li} W	(2*w_c) W = 1W	2.5	2.5
Real	Dual core 30% less powerful than single core i.e. w_c =0.7	2*(w_c) + 2 p_d =1.4+ 2 p_d W	w_s + P_{li} W	(2*w_c) W = 1.4 W	2.5	3.5
Theoretical	Quad core as powerful as single core	4* Pw_s = 4 W	1 + 3 P_{fi} W	0.25 + 0.25 + 0.25 + 0.25 = 1 W	1.25	1.25
Expected	Quad core 25% less powerful than single core (Expected) i.e. w_c = .25	4*(w_c) + 4 p_q =1+ 4 p_q W	w_s + 3 P_{li} W	4*(w_c) = 1 W	1.25	1.25
Real	Quad core 50% less powerful than single core i.e. w_c =.5	4*(w_c)) + p_q = 2 + 4 p_q W	w_s + 3 P_{li} W	(4*w_c) W = 2 W	1.25	2.5

The third column of the table depict the power consumed by different n BCE (single core, dual and quad core) processor to execute the sequential program. It can be observed that power consumption increases as the number of core is increased either by keeping r resources constant to create n-BCE multicore processor or by creating more than one n-BCE processor when all the cores are fully utilized.

The fourth column gives the amount of power consumed when a program is executed on multicore processor without parallelizing the program. The program is executed on a single core keeping other cores idle. If a non parallelized program (f=0) is executed on a multicore processor with n cores, only one core with r/n resources will execute the program while other (n-1) core will be idle consuming (n-1)*z unit of power where z is fraction of power that a core consume in idle state ($0 \leq z \leq 1$) [4]. If it is assumed that a core in active state consumes a power of 1 unit, i.e., the amount of power consumed by one core during the sequential computation phase is 1 unit, while the remaining ($n - 1$) cores consume ($n - 1$)*z units, during the sequential computation phase, the n core processor consumes $1 + (n - 1)*z$ units of power. In the parallel computation phase, n core processor consumes n units of power, because it takes ($1 - f$) and f/n to execute the sequential and parallel code respectively.

In general, the power consumed by the dual core and quad core processor where each core is n-BCE core is Pw_s + P_{fi} unit of power and Pw_s + 3 P_{fi} unit of power respectively. The power consumed by the dual core and quad core n-BCE processor is w_s + P_{li} unit of power and w_s+ 3P_{li} unit of power respectively where $w_{s=}$ w_c + p_d . It is observed that the power consumed by multicore processor is greater than Pw_s.

The fifth column gives the amount of power consumed by a program to execute on single and multicore (dual and quad core) after parallelizing the program. The power consumed by the dual core and quad core processor where each core is n-BCE core is Pw_s unit of power which is theoretically possible. The power consumed by the dual core and quad core n-BCE processor is $2*w_c$ unit of power and $4*w_c$ unit of power respectively. If it is possible to execute the program in half the execution time taken by single core processor by reducing the strength of cores by 50%, then the power consumption of a n-BCE single core processor and a n-BCE multicore processor will be equal. But, ideally the strength of the cores is not reduced by 50% thus increases the power consumption to execute the same program. It is observed that the power consumed by multicore processor is greater than $Pw_{s.}$ For example, if the power consumed by a program when executed for 5 unit time on a full blown single core processor Pw_s is 1. The energy consumed by the processor is 5 units. If the same program is executed on dual core processor whose strength is reduced by 30% compared to single core processor and if we assume the time taken is reduced to half i.e, 2.5 unit time. The power consumed w_c of each core is 0.7 unit and total power consumed is 1.4 W, but the energy spent is $2*(2.5*0.7) = 3.5$ units.

The performance per power (Perf/W), which represents the performance achievable at the same cooling capacity is based on the average power (*W*). This metric is reciprocal of energy, because the definition of *performance* is the reciprocal of execution time.

In other words, a sequential execution and its parallel execution version will consume the same amount of power only when the performance improvement through parallelization scales linearly. Otherwise $Pw_q > Pw_d > Pw_s$ to finish the same task.

Furthermore maximizing and balancing parallelization among cores is important, not only for higher performance but also for power supply efficiency and extended battery life.

1.2 State-of-the-Art of Exploiting Parallelism

All computer systems today, from embedded devices to petascale computing systems, are being developed using multicore processors.

Following are possible approaches available to exploit parallelism:

- Allow programmers to use parallel programming constructs to explicitly specify which parts of the program can run in parallel.
- Allow operating system (OS) to schedule different tasks on different cores.
- Allow hardware to extract parallelism and schedule them dynamically.
- Allow the compiler to extract parallelism and schedule them.

In first approach, developing and verifying an explicitly parallel program is expensive and doesn't scale with the number of cores [9].

In the second approach, the operating system realize each core as a separate processor and OS scheduler schedules coarse grain threads on to different cores. In the thread style approach, two explicit parallel primitives are independent unless an explicit communication primitive (for synchronization) are added to stress what is inside the original code. Further the multicore processor architecture differs from traditional multi processors in terms of having shared caches, memory controllers, smaller cache sizes available for each computational unit, and low communication latency between cores [18]. Owing to the architectural difference, it is desirable to extract fine grained thread and schedule them on to multiple cores instead of scheduling coarse grained thread as done in multi chip multiprocessing systems (SMP system).

In hardware-centric approach, detecting parallel execution opportunities and creating schedules for parallel regions dynamically by effectively utilizing all available resources is responsibility of the hardware [19]. This approach adds more circuits, which results in complex hardware implementations of algorithms such as branch prediction, instruction level parallelism detection, and register renaming. The hardware based approach work under heavy resources and time constraints.

In software-centric approach, a compiler analyzes the program for the possibilities of parallelism, identifies the code which could be executed in parallel and uses suitable scheduler to schedule the parallel constructs on to multiple cores. Using the various kind of dependency analysis, the compiler can identify the independent instructions that can run in parallel [20]. The compilation being offline one time activity, rigorous analysis to achieve optimal amount of code parallelization can be carried out.

1.3 Objectives and Contribution

The proposed research aims to provide compiler support to exploit parallelism by extracting fine grained threads from a sequential program and creating schedules for multiple cores.

The proposed work involves

- Parallel region formation or Extracting fine grain threads.
- Scheduling parallel regions or fine grain threads on to multiple cores. and
- Global register allocation.

The proposed work introduces two additional passes to the original flow of compiler: Fine grain thread extractor pass and scheduler pass. The fine grain thread extractor pass of the compiler splits the sequential program into parallel regions (fine grain threads) termed as sub-blocks. To facilitate global scheduling new data structure called sub-block dependency graph (SDG) is proposed. Efforts are made to reduce the compilation time for performing fine grain extraction pass. The sub-blocks are created such that they ensure spatial and temporal locality.

The fine grained threads can be scheduled using scheduler. One local scheduling heuristic, termed as Intra block scheduling and four global scheduling heuristics, termed as Inter block scheduling are proposed in the thesis. The scheduler ensures that the sub-blocks scheduled on different cores at same time will not communicate nor access same data, thus provide lock free synchronization. The schedulers are designed to perform power optimization.

The fine grained threads that are scheduled on to different cores need to be allocated registers from respective register file of the core on which they are scheduled. The register allocator perform the global register allocation on each list of sub-blocks dedicated to individual core. Two novel register allocation heuristics are proposed. The first approach proposes a register allocation technique which is performed after scheduling and the second approach integrates register allocation and scheduling pass to mitigate the phase order problem. In first heuristic, the interference graph is constructed incrementally by merging the sub-blocks to create hyper sub-blocks. Hyper sub-blocks are created before register allocation to ensure temporal locality by pushing maximum instructions on to a core for execution. Hyper sub-blocks also ensures that instructions will do zero spills (k-colorable) and will remain in cores private memory till it is commited without doing memory reference during execution.

1.4 Organization of the Thesis

A brief introduction to design philosophy behind multicore architecture, challenges and issues in multicore environment, state of art of exploiting parallelism, and objective of the thesis are discussed in the introductory chapter.

- In Chapter 2, we investigate several existing parallel architectures to understand how multicore is different from them. Several hardware and compiler support to exploit ILP (Instruction Level Parallelism) on these existing parallel architectures are presented and compared. An effort is made to understand the pros and cons of hardware and compiler approaches. A detailed survey on register allocation approaches is presented in the later part of this chapter.
- Chapter 3 describes the experimental framework used in this thesis. The framework includes a Compiler, Multicore architecture, the metrics used to evaluate the performance and the benchmark suites used in the proposed work. The phases of compiler are explained. A brief description on how to embed the SSA module in the given compiler is

explained. A short description of Transactional Memories (TM) which provide run time support in terms of lock free transactions is presented.

- Chapter 4 aims to provide the details of the support and optimizations achieved in the proposed work. Section 4.1 provide the detailed description of fine grain thread extractor module. Section 4.2 provide details of schedulers and Section 4.3 details the register allocation techniques. The optimizations includes creating power aware schedules and finding compile time efficient approaches.
- Chapter 5 discusses the parallel region formation techniques. Two different approaches are proposed to obtain disjoint sets (parallel regions). To facilitate global scheduling new data structure called sub-block dependency graph (SDG) is proposed and efficient technique to create it is discussed in detail.
- Chapter 6 explains the implementation details of local scheduling heuristics (Intra Block Scheduling) which creates schedules for the parallel regions within the basic blocks of CFG. Results in terms of speed-up, power consumption, performance per power and communication cost is presented at the end of this chapter.
- Chapter 7 introduces four global scheduling heuristics (Inter Block Scheduling) which schedules the parallel regions formed across the basic block of the CFG. The brief discussion on merits and demerits of each heuristics are presented by comparing the results obtained by them. The results obtained by Inter block scheduling is also compared with the results obtained by Intra block scheduling technique.
- In Chapter 8, two register allocation techniques for multicore architecture are proposed. The first approach proposes a register allocation technique after scheduling and the second approach introduces a technique of integrated register allocation and scheduling approach to mitigate the phase order problem. The results obtained by the normal register allocation approach and integrated approach is compared and presented at the end of the chapter.
- Chapter 9 concludes the thesis by summarizing the achievements of the work, providing limitations and suggests future direction.

  Appendix A, Appendix B, List of references and List of publications by author is appended to chapter 9.

CHAPTER 02 *Literature Survey*

To achieve high performance computing, a single core processor with parallel processing features were developed during 1975–2000 before multicore architecture was introduced by IBM in 2001. These parallel architectures either had multiple instruction processing units or multiple functional units. As computer architecture started becoming more complex, the compiler technology has also equally became an important factor. The success of each innovation in computer architecture is dependent on the ability of compiler technology to generate efficient code for these architectures. Parallelism has become one of the distinguishing factor in the design of high-performance computers. Parallelism comes in different form, namely instruction level parallelism (ILP), Task / Thread level parallelism (TLP), Memory level parallelism etc. A compiler was used by the parallel architectures to exploit parallelism as required by them to squeeze more performance.

This chapter discusses the relationship between parallel architectures, Instruction Level Parallelism (ILP) extraction techniques and compiler support to exploit ILP for corresponding architecture. Several existing parallel architectures such as pipeline, VLIW, and superscalar architectures are investigated to understand how multicore is different from them. Several techniques in both form, dynamic (hardware) and static (compiler) support to exploit ILP on these existing parallel architectures are presented and compared. In section (2.4) a detailed survey on register allocation approaches is presented and examines the register allocation requirement for multicore architectures.

2.1 Parallel Architecture, ILP & Compiler

The principle behind RISC architecture is to move the architecture boundary closer to the hardware, exposing key performance features to the compiler. By doing so, it can take advantage of the compiler by off-loading the task like choreographing complex instructions from the hardware to compiler, to get high performance processor. Some of the new generation of the microprocessors have implemented branch prediction, ILP detection, register allocation or renaming and hazard detection logic in hardware to achieve ILP and faster execution.

The analysis at compile time can simplify and eliminate many of the complex algorithms in the hardware. Some architecture such as Power4 [21], Cyclops [22], RAW [23][24] and TRIPS [25][26] aims to maximally utilize the compiler by fully exposing the hardware and giving control to the software systems. Furthermore, the rigorous compiler-based analysis can lead to improved optimizations as compared to hardware-based approaches which work under heavy resource and time constraints. The current day compilers can analyze the complete program to infer detailed information about ILP in a given program code.

Instruction level parallelism (ILP) is a technique used to speed up the execution of code by allowing parallel execution of sequence of instructions derived from a sequential program [27]. The exploitation of ILP in a code is majorly hampered by conditional branch instructions and dependent instructions. The dependency analysis can be carried out to identify the set of independent instructions that can be executed in parallel. The instruction dependency is of three types, namely the name dependency, the control dependency and the data dependency. There are two types of name dependencies, Write after Write dependency (WAW) or anti dependency and Write after Read dependency (WAR) also known as output dependency. The name dependency can be eliminated by register renaming. Dynamic register renaming (by hardware) can eliminate WAW and WAR dependencies. But when an intermediate representation of program in static single assignment (SSA) form is used, WAW and WAR dependencies are removed without any need of hardware [28]. SSA form is an intermediate representation of a program in which each variable is defined only once. The control dependencies can be removed by using the hardware to predict conditional branches. Read after Write (RAW) dependency also known as true dependency falls under the data dependency category. It can be removed at run time using data collapsing [29] and re-association [30] technique. These techniques require specialized hardware elements. The compilers can be used for carrying out in-depth code analysis to determine the data dependency. Compiler driven optimizations are likely to significantly improve the execution performance of a processor.

To exploit the instruction level parallelism, first in-depth data dependency analysis is carried out. This analysis is used for segregating dependent and independent set of instructions for scheduling & resource binding. The advancement in the field of VLSI technology has led to design of parallel architecture, and the compiler is used for exploiting ILP on such architectures [31]. The nature of ILP support offered by compiler is heavily dependent on the architecture and varies for different architecture. The interplay between compiler support and available architectural features is shown in Figure 2. Compiler developed for sequeintal architectures such as superscalar architectures, does not perform any machine level optimizations and does not convey any explicit information regarding parallelsim, special hardware performs machine specific optimizations.

In Dependence and Indipendent architectures such VLIW and Horizon architectures, the responsibility is of machin level optimization is shared between compiler and hardware. Compiler explicily indicates the dependences that exist between operations.

In fully indipendent architectures such as RAW architecture, compiler will be fully aware of features of the processor and will take full responsibility of machine level optimization. The adventage of these type of architectures is that, execution time and power is saved.

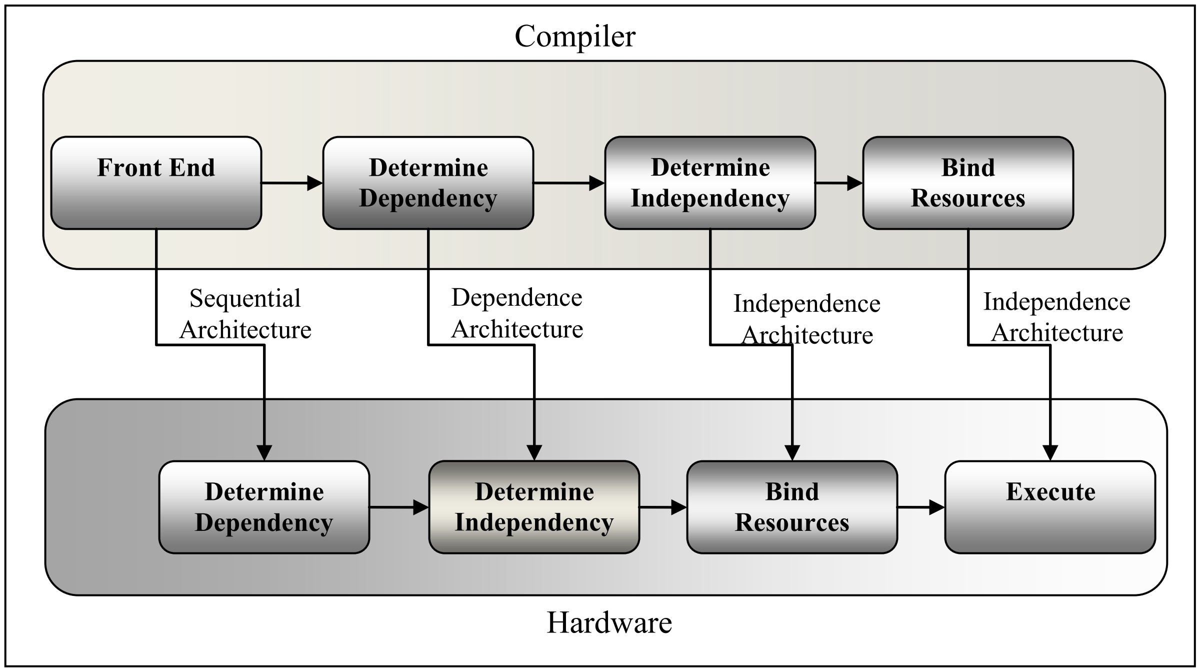


Figure 2. Compiler vs Hardware Support for Exploiting ILP

The hardware approach for achieving ILP is being able to execute multiple instructions simultaneously either by pipelining the instructions or by providing multiple execution units. Pipelined processor, VLIW (Very Long Instruction Word) and super-scalar processors exploit ILP to improve execution time. In pipelined processor, a task is broken into stages, and stages are executed on different (shared) processing units by overlapping the execution of instructions in time [32]. The performance resulting from pipelining is expected to increase with increase in pipeline stages. However, pipelined operations are required to be continuous without interruption throughout the program execution. Unfortunately, the processor sometimes stalls as a result of data dependency and branch instructions. RISC solution to this problem is code reordering [33]. The task of code reordering is generally left to the compiler, which recognizes data dependencies and attempts to minimize performance stalls by reordering the program instructions.

VLIW processor follows the static scheduling. VLIW issues one long instruction per cycle. Each long instruction consists of many tightly coupled independent operations. These independent operations are simultaneously processed by suitable execution units in a small and statically predictable number of cycles. The task of grouping independent operations into a long instruction is done by a compiler [34]. The major drawback with VLIW is that it uses the fixed number of instructions. The availability of multiple execution units is not utilized completely, because the execution unit which has completed its processing will be idle until all the execution units have completed their processing. Super-scalar processor overcomes the drawback of VLIW by working on variable number of instructions using simultaneous multithreading, where independent threads will run in parallel [35]. The major drawback with the super-scalar processor is that all the execution units share the same memory leading to more register spilling, and race condition due to limited availability of registers.

The multicluster VLIW embedded processor is made up of multiple small processing elements (PEs) [36][37][38]. These PE's are individual groups designed by decentralizing the computing resources to improve the scalability problem. Each tightly interconnected PE's help to reduce the communication cost & power. The instructions partitioned by compiler analysis are executed in parallel on these PEs [39]. The main difference between today's multicore processors and multicluster VLIW processor is that later has shared data cache, while each core in multicore processor will have private caches. The compiler should also be aware of data which are brought into private cache of the core.

VLIW and superscalar machines, both benefit from code reordering. In VLIW, all dependencies are checked during compile time, and the search for independent instructions and scheduling is done exclusively by the compiler. The hardware has no responsibility on the final scheduling. On the other hand, superscalar machines depend on hardware for scheduling the instructions. But it is accepted that compiler techniques for exploiting parallelism must be used in superscalar machines to achieve better performance.

2.2 Background of Instruction Scheduling

In case of Superscalar and VLIW machine, the scheduling of instruction is dependent on identification of set of independent instructions that can be executed in parallel. The scheduler only addresses the issues associated with temporal parallelism leading to exploitation of ILP but it may increase register pressure [40]. These schedulers do not take care of spatial issues as superscalar and VLIW processors exchange the shared/dependent operands through shared register file which is absent in multicore system. For the pipeline based machines, scheduler reorders instructions to minimizes pipeline stalls. The reordering of the instructions should not change the set of operations performed and should make sure that interfering operations are performed in order.

In the past, researchers have proposed several instruction scheduling techniques which includes List scheduling, Trace scheduling, Superblock scheduling and Hyper block scheduling. All these scheduling techniques can be classified based on the nature of the control flow graph used, i.e. whether it uses multiple or single basic blocks, and whether it is cyclic or acyclic control flow graph.

The scheduler that schedules single acyclic basic block is known as local scheduling. List scheduling is an example of local scheduling [41] and is based on highest level priority scheme. Trace scheduling, superblock and hyper-block scheduling are global scheduling techniques that work on regions known as traces which consists of contiguous set of basic blocks [42]. Trace scheduling combines the most common trace of basic blocks and schedule them as a single block [43]. Superblock scheduling is same as trace scheduling without side entrances [44]. Hyper-block scheduling combines basic blocks obtained from multiple paths of control flow graph [45].
In run-time scheduling, an instruction is issued after it is decoded and when its operands are available [46]. The run-time scheduling mechanisms exhibit adaptive behavior which leads to higher degree of load balancing. The run-time scheduling policies incur high run time overhead which may lead to degradation of execution performance. The logic to make decision at run time should be simple and constant time heuristics, otherwise it leads to expensive and complex hardware design which requires relatively large amount of silicon area. The complex hardware in turn results in increased power consumption. The advantage of compile time scheduling over the run-time scheduling is that it can carry out rigorous dependency analysis. The complexity of the

scheduling techniques will affect the compile time of a program but has no adverse impact on its execution time.

Scheduling techniques for various parallel architectures are summarized in Table II illustrating the advantage, drawbacks and algorithm complexity of these techniques.

TABLE II. SCHEDULING TECHNIQUES FOR PARALLEL ARCHITECTURE

Scheduling Technique	**Architectur e**	**Type**	**Details**	**Complexity**
Basic Block Scheduling [47]	Pipeline	Static	Topological sort of Dependence graph. Simple and easy to implement. Restricted to a single block.	O(N log N)
Region Scheduling [48]	Pipeline	Static	Creation of regions from blocks and then topologically sorting the regions. Inter block scheduling is made possible. Operations within a loop cannot overlap with those of another loop.	O(N^2)
Gibbons-Muchnick method [49]	Pipeline	Static	Creation of DAG. Then choosing the instruction to be scheduled according to heuristics such as:- the node with the max no of children or which interlocks with its children or which is on the longest path from the leaves. Heuristic approach makes scheduling simple. Dependency DAG's can be used for other code optimizations. Deadlocks are prevented. Much of the hazard detection is assumed to be done by the hardware to make things simple. Also, scheduling across basic blocks is not considered.	O(N^2)
Bernstein's method [50]	Pipeline	Static	First a level is assigned to each instruction. Next a list scheduling is performed in decreasing order of priority. Nodes on the critical path are assigned higher priorities and therefore scheduled first. Structural hazards cannot be avoided as the DAG is not weighted i.e. it does not take into account the latency of each operation.	O(Nlog N)

Shieh-Papachristou method [51][52]	Pipeline	Static	Construction of priority list using several heuristics and then assigning time slots based on the priorities. Using multiple heuristics reduces the probability of choosing a wrong node for scheduling. The greedy heuristics used can schedule two floating point operations in consecutive cycles even when the processor is not pipelined.	O(N + E)
Superblock Scheduling [53]	Pipeline	Static	Involves finding the most frequently executed path using trace selection. Then superblocks are formed from these traces and DAG is formed, after which list scheduling is performed. Reduces the complexity associated with side entrances by removing them. Also, it focuses on the frequently executed paths. Cannot deal with the situation when different execution paths have equal frequencies of execution.	O(N*E)
The Scoreboard [54][55]	Pipeline	Dynamic	Used multiple execution units for out of order execution. A centralized control unit called the scoreboard is responsible for instruction issue and execution, including the detection of hazards. Enables out of order execution. Allowed instructions behind stalls to proceed. Structural hazards stall the pipeline. Limited to instructions in basic block.	
Tomasulo Algorithm [56]	Pipeline	Dynamic	Two floating point units are used – add and multiply/divide. Buffers called reservation stations are used for fetching and storing instruction operands. The result of an operation is stored at reservation stations to remove WAW and WAR hazards. Register renaming removes structural hazards. Limited to instructions in a basic block. A common data bus limits the amount of data transfer.	

List Scheduling [57]	VLIW and Superscalar (but mainly for superscalar)	Static	It finds ILP within a single basic block and fills data stalls (if any) by instructions in other basic blocks.	***O(N)***
Trace Scheduling [58]	VLIW	Static	It exploits ILP across basic block boundaries and tries to locate the most frequently executed path called *trace* by preserving program semantics (book-keeping).	***O(N^2)***
Software Pipelining [59]	VLIW	Static	It tries to find the maximum ILP through the loop unrolling and scheduling the iterations of the loop every initiation interval.	***O(N^2)***
Enhanced Modulo Scheduling [60]	VLIW and Superscalar	Static	It schedules the loops by maintaining an optimal value of initiation interval.	***O(N^2)***
Speculative Execution [61][62]	VLIW and Superscalar	Static	It tries to speculatively execute instructions that are moved upward of conditional branches.	***O(N^2)***
Superblock Scheduling [63]	VLIW and Superscalar	Static	It is derived from trace scheduling. It schedules the instructions from most frequently executed and optimized ***superblock*** (a trace with no side entrance is called a superblock).	***O(N^2)***

Next section explains the motivation to carry out the research in the area of multicore architecture.

2.3 Scheduler Requirement for Multicore Architecture

i. All the existing techniques find it difficult to make good decision on scheduling because, scheduling algorithms are strongly dependent on the machine model for which they are developed. The instruction scheduling techniques are NP-complete and follow heuristics. Some of the heuristic/ practices are loop transformations, static branch prediction, speculative code motion, predicated execution, software pipelining, and clustering. Different heuristics work well with different types of graph.

ii. In the existing local scheduling techniques, ILP is achieved by scheduling one instruction at a time on multiple execution units. To perform this task, a critical path of instructions is created by analysing the dependencies. The instruction with lengthy critical path is scheduled first to enable other instructions to get scheduled. This technique cannot be applied on multi-core environments because two dependent instructions may get scheduled on different cores resulting in increased communication latency. The ILP on multicore architecture can be fully exploited only if all dependent instructions are scheduled on to same core.

iii. The existing global scheduling techniques work on multiple basic blocks of a CFG and try to group them based on the dependency analysis. Since there are three dependencies to look for, the output of these techniques were discouraging for most of the applications. The programs in static single assignment (SSA) have proved useful by eliminating false dependencies in traditional code. Removing false dependencies allows more flexibility in scheduling since data independent operations can move close to each other during instruction scheduling. Along with simplifying the dependency analysis among the instructions, SSA form programs gives solutions to the class of NP-complete problems like register allocation and enables various optimizations [64][65]. The proposed work is performed on SSA form program, which forms the backbone of further analysis.

iv. In most of the traditional instruction scheduling algorithms the goal is to improve performance in terms of execution time by increasing the amount of instruction-level parallelism in program code. Since communication between distant computing resources may invite delays, instruction scheduler is expected to take care of spatial problem along with temporal problem in multicore environment. The instruction scheduler needs to partition instructions across the computing resources. Based on the parallel schedule generated by compiler, the power consumption may vary [66]. Power reduction without impeding the speedup is an important scheduling constraint for parallel architecture. The proposed work involves power-aware scheduling strategies which minimizes the switching activities between instructions and use reduced number of cores to achieve performance per power.

2.4 Survey on Register Allocation

Register allocation is a crucial phase of compilation. It maps unbounded number of variables of a program to a fixed number of physical registers of a processor. Values stay in registers as long as variables are live, In general, register allocation problem is NP-complete from the fact that number of registers available is small and some of them are special purpose registers. Due to limited number of available registers, the register allocation to all variables will not be possible, and hence are required to be stored in memory. These variables are called spilled variables. The cost of spilling is minimized by spilling the least frequently used variables. The commonly used register allocation approaches are shown in Figure 3.

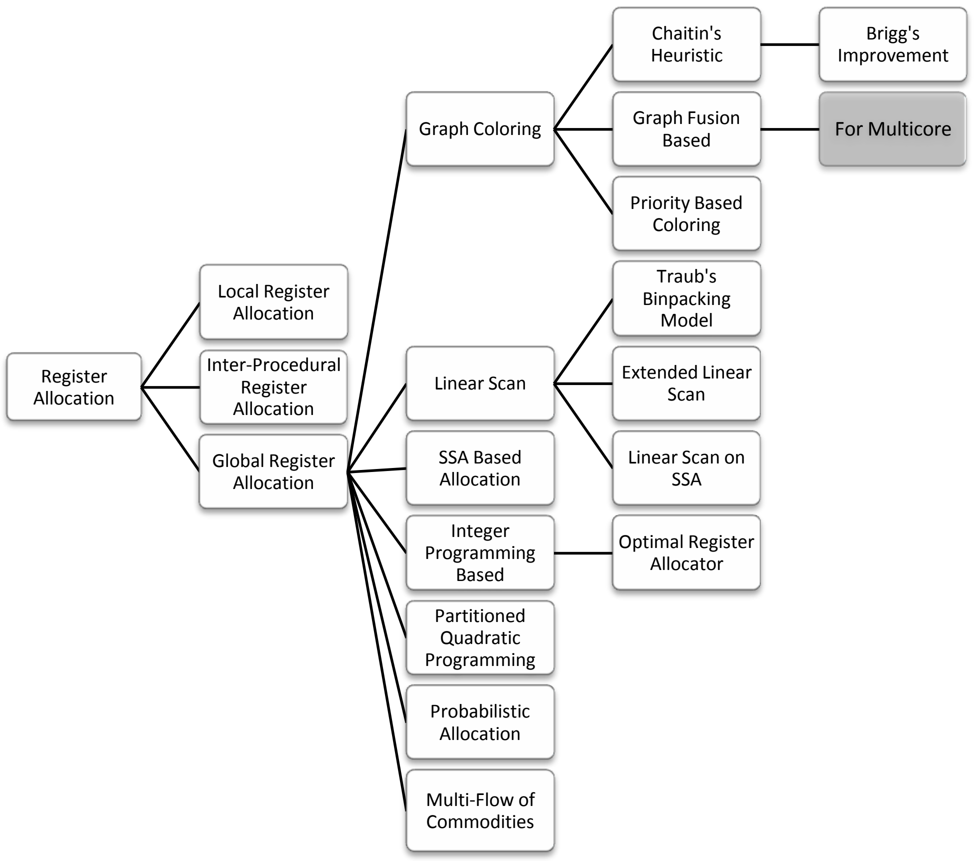


Figure 3. Register Allocation Approaches

Most of the register allocation algorithms assume that the CPU has regular register files and these algorithms fail to adopt themselves for irregular architectures. Several solutions have been proposed for irregular architectures, but without considering the specific implementation details, it is difficult to achieve optimal register allocation [67][68][69]. The following section discusses the various register allocation approaches for single core processors [70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82]. In case of multicore processor, each core of the processor has individual register file and optimal register allocation is of utmost importance. Multicore architecture requires new strategies for register allocation. A brief summary of comparison of the various register allocation algorithms is given in the Table II1. The Pros and Cons of these register allocation approaches are given in Table IV.

Static single assignment (SSA) is an improvement on the idea of def-use chains. The advantages of SSA form programs for register allocation are:

i. Coloring the interference graph of SSA form can be accomplished in polynomial time [83,84]. The interference graph of SSA form program is a chordal graph and it inherits all the properties such as [85].
    - Chordal graphs are perfect, whose chromatic number is equal to size of the maximal clique.
    - Chordal graphs have a simplical vertex which facilitates perfect elimination order (PEO). PEO assists the simplification process of interference graph during coloring.
    - The key to good spill-code generation lies in splitting the live-range of a variable at the right places. Splitting is obvious in SSA form program as every variable has a single contiguous live range. This reduces the register pressure.

ii. In case of register allocation using linear scan, the lifetime intervals can be constructed easily from SSA form [82].

TABLE III. COMPARISON OF REGISTER ALLOCATION APPROACHES

Allocation Approach	**Interference Graph Notion**	**Region Notion**	**Live Range Notion**	**Spilling Approach**	**Design Paradigm**	**Complexity**
Chaitin’s	Per Function	Basic Block, Super Block	Program Points	Aggressive	Approximation	O(|V|* log(V)) V = num of live-ranges
Brigg’s	Per Function	Basic Block, Super Block	Program Points	Delayed; Optimistic	Approximation	O(|V|* log(V)) V = num of live-ranges

Priority Based	Per Function	Basic Block, Super Block	Set of Basic Blocks	Lower priority, constrained live ranges; Incrementally Add live-ranges till colourable.	Greedy	O(r * (V-r)) r = no of registers V = no of live-ranges

Graph Fusion	Per Region	Basic Block, Super Block, Function	Program Points	Delayed; Optimistic	Combine-and-Conquer	O(f * (V + E))
						f = #fusion-ops
						V = #liverange
						E = #interference Edges

Linear Scan	Per Function	Basic Block, Super Block	Program Point: relaxed, with first def/use to last def/use, without considering discontinuity.	Aggressive	Greedy	O(V * E)
						
						V = #liveIntervals
						E = #interference Edges

Traub's Bin-Packing Linear Scan	Per Function	Basic Block, Super Block	Program Points: strict, with first def/use to last def/use, but, considering discontinuity.	Spills only if the current live range does not fits into an existing live-range's hole.	Greedy	O(V * E)
						
						V = #liveIntervals
						E = #interference edges

Multi-Flow of Commodities	Per Function	Basic Block, Super Block	Program Points	Takes spill decisions at a point subject to minimizing the overall cost of the flow graph.	Incremental, Heuristic Based, Network Flow.	O(|V| + |E|)
						V = #liverange
						E = #interference edges

Integer Programming	Per Function	Basic Block, Super Block	Program Points	Takes spill decisions at a point subject to minimizing the overall cost of variable usage and imposed constraints.	Linear Programming	$O(V^3)$ V = #liverange

Partitioned Quadratic Programming	Per Function	Basic Block, Super Block	Program Points	Takes spill decisions at a point subject to minimizing the overall cost of variable usage and imposed constraints.	Numerical Programming	$O(V * \|k\|^3)$
						
						V = #variables
						k = #registers

TABLE IV. PROS AND CONS OF VARIOUS REGISTER ALLOCATION APPROACHES

Allocation Approach	**Pros**	**Cons**
Chaitin's	Simple and intuitive	In case a live range is spilled due to lack of registers at a program point, all uses of that live range go through memory even though some parts of the live range could have been allocated a register.
Brigg's	May alleviate the problem with Chaitin's approach for certain programs.	Does not eliminate the above problem with Chaitin's approach.
Priority Based	Attempts to assign registers to the most important live ranges and to spill the least important ones if necessary; maintains the simplicity of the graph colouring based allocation approaches.	Takes neither execution frequency nor program structure into account when splitting live ranges, and there is no guarantee that splitting points do not end up along frequently executed edges raising code execution time.
Graph Fusion	For small programs, provides results identical to Chaitin's, in lesser time usually; For programs with huge register pressure, uses profile information to produce better register assignment, which cannot be done in other graph colouring or linear scan algorithms.	May result in partial redundancies.
Linear Scan	Simple, Faster than graph colouring based approaches, used with JIT compilation.	Non-Optimal; Does not handle "holes" in live-ranges.
Extended Linear Scan	Guarantees minimal number of Register usage; Simple; Faster than graph colouring based approaches; used with JIT compilation.	May insert too many copy-swap instructions
Traub's Bin Packing Linear Scan	Simple, Faster than graph colouring based approaches, Used with JIT, Handles "holes" in live-ranges, Better allocation than simple Linear Scan.	Non-Optimal.

SSA Based Approaches	Colouring & Spilling can be de-coupled, Colouring can be done in polynomial time, Lower register pressure.	Additional time and complexities (lost copies etc) arising out of use of SSA form.
Integer Programming Based	Powerful design paradigm; Produces very good quality code; Separation of spilling & code generation makes process faster; Reduces code size.	Runs into exponential time in worst case (Linear Programming is NP-Complete); May produce large number of “move” instructions; Complex to represent Interference graph as a set of 0-1 linear equations.
Partitioned Quadratic Programming	Can find optimal allocation in 97.5% of cases; Heuristics can be used in rest of the cases; runs in polynomial time.	PQP is NP-Complete in general; May not be able to handle non-disjoint register aliases; Complex to represent Interference graph as a set of quadratic equations.
Multi-Flow of Commodities	Produces reduced size programs than graph-coloring based approaches; Quickly finds initial allocation using heuristics.	May take a long time to get to the most optimal register allocation.

Instruction scheduling and register allocation phases have received wide attention in industrial and academic research, but are generally considered as separate problems. Traditionally, instruction scheduling and register allocation are performed independently. Either scheduling or register allocation can be performed first followed by the other. These two phases have conflicting goals and work in an opposing manner. Instruction scheduling aims at keeping the functional units busy by executing maximum number of parallel instructions in a short period of time. This requires a large number of values to be held in registers causing numerous spills. On the contrary, the register allocator aims at keeping the register pressure optimum by holding a small number of values in registers for a long period, leading to decreased utilization of the CPU.

Phase ordering problem has severe impact on code optimization. At times to exploit the ILP, instruction scheduling phase precedes the register allocation. This approach sometimes increases the register pressure. Alternatively in order to achieve efficient utilization of register file, register allocation phase can be carried out before the instruction scheduling phase. Though this approach will result in efficient register utilization but during the instruction scheduling phase, it may create empty slot schedule causing increased execution time. Studies on the phase ordering problem have tried to combine the instruction scheduling and register allocation phases to address the issues related to register spilling and loss of ILP [86][87].

Integrated pre-pass scheduling (IPS) combines a pre-pass scheduler with a liveness analysis to estimate register pressure at the beginning of each basic block in the program [88]. When a variable is defined, it increases register pressure by getting allotted a register and when it is done with its last use, the register pressure decreases by freeing that register. The register pressure is monitored continuously and when the pressure crosses a threshold, IPS prefers to shorten live ranges resulting in spills.

The parallel interference graph approach uses an extended interference graph to detect excessive register demands and guide schedule sensitive register allocation (PIR) [89]. The reduction in register demands is achieved through live range spilling.

The unified resource allocation (URSA) method is based on register reuse direct acyclic graph (DAG) [90]. Edges in a register reuse DAG connect two instructions if the target instruction can reuse a register freed by the source. It is based on the measure and reduce paradigm. Groups of instructions that use too many registers if scheduled in parallel are identified. These are called excessive sets and they are then used to drive reductions of the excessive demands for resources. Live range splitting is used for reducing the register pressure. Various groups have implemented code generators integrating optimal instruction selection, instruction scheduling and register allocation, based on formulations such as integer linear programming.

2.5 Recent Developments

The multicore processors offer abundant computing resources offering opportunity to exploit ILP. The thesis, presents the techniques for exploiting the fine grained parallelism for multicore processor using compiler. This is the first attempt of its kind to exploit a fine grain parallelization using compiler.

Currently the research community is trying to counter the challenges either by designing new schedulers (both dynamic & static), Data access partitioning to extract fine grain threads, pre-fetching data on to private caches of the core, and load balancing. These work are being done in parallel to the work presented in this thesis and are summarized in Table V.

TABLE V. RECENT DEVELOPMENT IN THE AREA OF MULTICORE

Approach	Type	Year	Description
Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine [91]	Static	1998	This work was proposed as part of RAW project. The RAWCC compiler was developed to compile general purpose sequential programs to distributed RAW architecture. The scheduler was used to assign instructions in basic block and data belong to that instruction to processing unit. The technique to exploit ILP in this work is an improved version of the concept what is used for pipelined processors.
Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications [92]	Dynamic	2007	Proposes a multicore architecture, referred to as Voltron, that extends traditional multicore systems in two ways. First, it provides a dual-mode scalar operand network to enable efficient inter-core communication and lightweight synchronization. Second, Voltron can organize the cores for execution in either coupled or decoupled mode. In coupled mode, the cores execute multiple instruction streams in lock-step to collectively function as a wide-issue VLIW. In decoupled mode, the cores execute a set of fine-grain communicating threads extracted by the compiler.
Data Access Partitioning for Fine grain Parallelism on Multicore Architectures [93]	Static	2007	The work aims to reduce dispersal of data accesses across the cores. A profile-guided method is proposed for partitioning memory accesses across distributed data caches. The profiler determines affinity relationships between memory accesses and working set characteristics of individual memory operation in program. The compiler uses the profiled information to perform program-level partition of the memory operations to divide the memory accesses across the data caches. As a result, the data accesses are proactively dispersed to reduce memory stalls.

Fine-grain Parallelism using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function [94]	Static	2009	This work focuses on exploiting fine-grain parallelism in Mr Bayes, a well-known Bioinformatics application as loop-level parallelism is a common characteristic of such scientific applications. Three different existing architectures such as general purpose multicore processor, Cell/BE, and Graphics Processor Units (GPU) systems are analyzed in terms of execution performance, scalability and programmability.
Compiler Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors [95]	Static	2009	Proposes an automatic parallelization approach for transforming input affine sequential codes into efficient parallel codes such as OpenMP, that can be executed on a multi-core system in a load-balanced manner. This approach employs a compile-time technique that enables dynamic extraction of inter-tile dependences at run-time, and dynamic scheduling of the parallel tiles on the processor cores for improved scalable execution.
Compiler-assisted Data Distribution for Chip Multiprocessors [96]	Static	2010	Presents a compiler-based approach for analyzing data access behavior in multi-threaded applications to mitigate the effect of data access latency. As in traditional data access analyses such as reuse, dependence and locality analysis which focus on affine array subscript patterns in loop nests of a single threaded application. They propose a technique to find the relationships of memory locations accessed by different loop iterations in a parallel programming context.
Resource-Aware Compiler Perfecting for Many-Cores [97]	Static	2010	Try to address the memory level parallelism issues thrown by the shared caches in multi and many core environment. Propose and evaluate a compiler loop pre-fetching algorithm targeted at many-core architectures and is aware of the number of simultaneous pre-fetches supported.
Dynamic Scheduler for Multi-core Systems [98]	Dynamic	2010	This paper propose a dynamic scheduling algorithm in which the scheduler resides on all cores of a multi-core processor and accesses a shared Task Data Structure (TDS) to pick up ready-to-execute tasks. In this method the processor has the onus of picking up tasks whenever it is idle.

Contention-Aware Scheduling on Multicore Systems [99]	Static	2010	This work investigates how and to what extent contention for shared resource can be mitigated via thread scheduling. The work identifies a classification scheme for threads to determine how they affect each other when competing for shared resources. The classification scheme, along with contention for cache space addresses the contention for other shared resources, such as the memory controller, memory bus and pre-fetching hardware.
Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications [100]	Static	2011	This work is carried out by Compilers Creating Custom Processors (CCCP) group in The University of Michigan. This dissertation tackles many challenges faced in automatic parallelization of sequential applications. The first phase of the work identifies the parallelizable portion in the program and converts it to parallel version. In the second phase they propose a runtime system STMLite to monitor the parallelized program behavior.
An automatic parallel code generation tool for data translation of non-uniform FFT (NuFFT) application for multicore processors is proposed in this paper [101]	Static	2012	Two scalable parallelization strategies, namely, the source-driven parallelization and the target-driven parallelization is used. To improve data locality while trying to balance workloads across the cores, equally sized geometric tiling and binning strategies are employed. This tool also consists of a code generator and a code optimizer for the data translation.
Compilers for Low Power with Design Patterns on Embedded Multicore Systems [102]	Static	2013	In this work case studies are presented to investigate compilers for low power with parallel design patterns on embedded multicore systems. Two major parallel design patterns: Pipe & Filter and Map Reduce with Iterator are evaluated. Authors has attempted to devise power optimization schemes in compilers by exploiting the opportunities of the recurring patterns of embedded multicore programs. In all two cases of the patterns investigated, the common recurring patterns of programs are exploited to seek the opportunity for compiler optimizations for low power. Proposed optimization schemes are rate-based optimization for Pipe & Filter pattern and early-exit power optimization for Map Reduce with Iterator pattern.

Increasing Off-Chip Bandwidth in Multi-Core Processors with Switchable Pins [103]	Dynamic	2014	This work address the bottleneck due to slow memory accesses by increasing off-chip memory bandwidth by enabling more memory channels. The main contributions in the work includes, devising a memory controller that can dynamically increase the off-chip bandwidth at the cost of a lower core frequency and a switchable pin design which can convert a power pin to a signal pin or the other way around. The switching policy is dynamic which will identify the memory intensive phases. This switches the system to prioritize memory bandwidth or core performance according to the identified phase.

2.6 Conclusion

Exploiting ILP from a sequential program is a combined effort of hardware and compiler. This chapter provides an overview of various parallel computer architectures and compiler for these architectures. The parallel architectures include pipelined processors, VLIW, superscalar, and clustered VLIW processors.

The computer architectects of these parallel architectures were aware of support offered by compiler at different levels to exploit ILP, i.e. compiler support is one of the design issues to be considered while designing these parallel architectures.

To generate high quality code for these architectures scheduling and register allocation need to be efficiently implemented along with various analysis, optimization and transformations of the program written in high level language. A detailed study of scheduling techniques and various register allocation techniques are studied and presented in this chapter. The outcome is the knowledge required to understand the research gap in multicore and many core architecture.

CHAPTER 03 *Experimental Framework*

In this chapter a generic compiler frame work for multicore architecture has been proposed with a view to exploit instruction level parallelism. A sample benchmark program is used for analysis of the performance of proposed framework. The speedup, power consumption, performance per power and communication cost is used as performance metric. The proposed frame work is generic and is independent of architecture.

The experimental setup uses Jackcc an optimizing C Compiler that generates code using Jackal ISA. Without loss of generality, for the ease of computation, it is assumed that each core of the multicore processor takes on an average one cycle for executing an instruction. Each core of multi core processor is considered to be equivalent to the single core processor. It is also assumed that there is no context switch while executing a parallel region on a multicore processor.

3.1 Multicore Architecture

The target multicore architecture model used is a fine grained architecture. This architecture exposes the low level details of hardware to compiler. The architecture supports minimal set of mechanisms in the hardware and these mechanisms are fully exposed to runtime software environment and compiler. The runtime system manages mechanisms historically managed by hardware, and compiler has responsibility of managing issues like resource allocation, extracting parallel constructs or fine grain threads for different cores, and creating schedules.

The multicore environment has multiple interconnected tiles and on each tile there can be one RISC processor or core as shown in Figure 4 & Figure 5. Each core has instruction memory, data memory, PC, functional units, register files, configurable logic and source clock. FIFO is used for communication. The register files are distributed, eliminating the small register name space problem. The cores are assumed to be homogeneous. Such architectures can be seen in Power4 [21], Cyclops [22], RAW [23][24] and TRIPS [25][26] architecture.

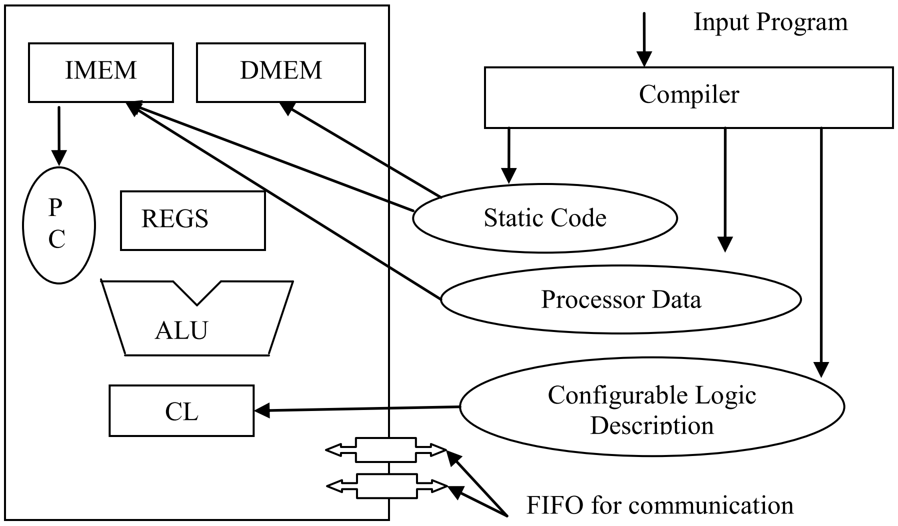


Figure 4. Compiler Generated Code and Their Relation with Architecture (Core/Processor)

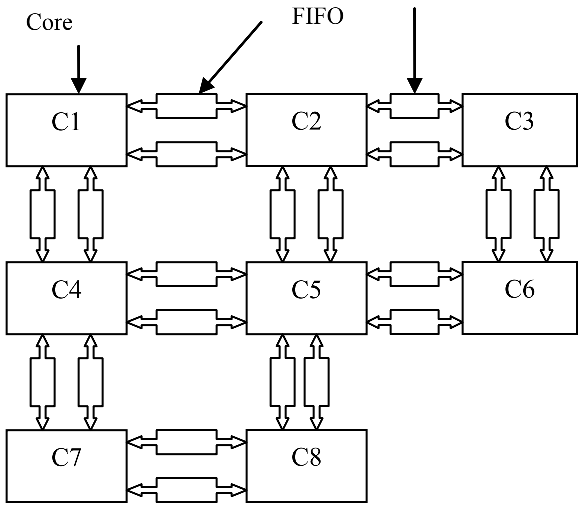


Figure 5. On Chip Multiprocessors (Cores) Interconnections

3.2 Compiler

The proposed work uses Jackcc Compiler [102]. This is an open source compiler developed at university of Virginia. The work flow of the compiler is shown in Figure 6. The compilation process is divided into manageable units called phase. The front end module of the compiler takes the source code as input and produces the DAG. The DAG2CFG module extracts the quads and then it forms the CFG consisting of basic block. The basic block in CFG of Jackcc is called *Arena*, and instruction inside the block is called *Quad*.

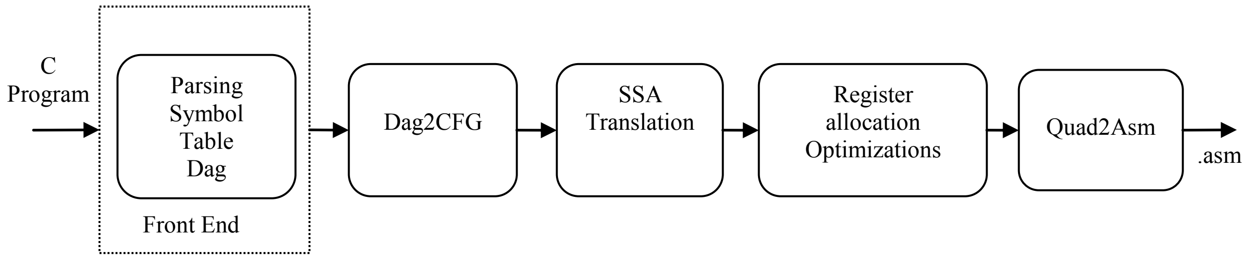


Figure 6. Original Flow of Compiler

Instructions are in SSA form. The original Jackcc compiler used SAME function instead of implementing Φ functions. The register allocator places two live ranges in the same physical register. A SSA conversion module has been integrated within in Jackcc compiler. The process of converting a Non-SSA form program to SSA form program has two steps and these steps are shown in Figure 7.

Step 1: Placing Phi (ϕ) statements by computing iterated dominance frontier.

Step2: Renaming variables in original program and Phi (ϕ) functions, using dominator tree and rename stack to keep track of the current names.

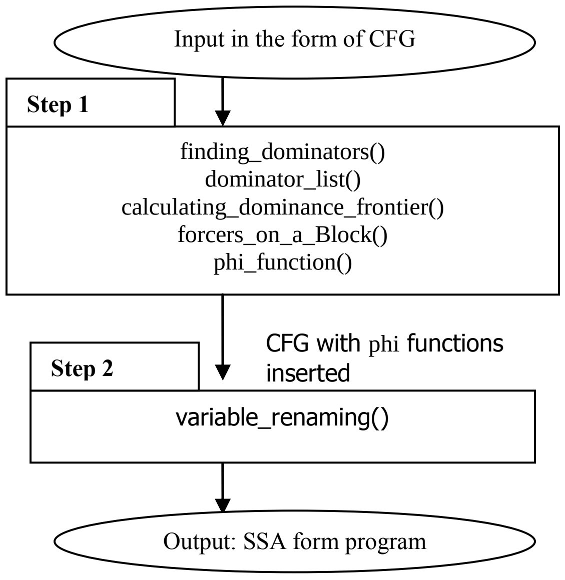


Figure 7. Sequence of Functional Call in Compiler to Generate SSA form Program

Register allocation is achieved by color_graph module of Jackcc compiler which is based on Chaitin's register allocation approach. The various phases of color_graph module is shown in Figure 8.

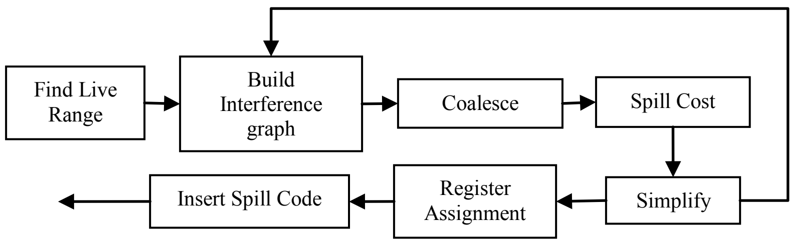


Figure 8. Register Allocation Framework

Live range of the variable is computed by performing liveness analysis using def-use chains and each live range's are numbered uniquely. An interference graph is constructed with the help of live ranges. The resulting interference graph may not be k-colorable. Coalescing and simplification are done to make this interference graph k-colorable. The Coalesce phase removes redundant copy instructions by combining the sources of the target live ranges. Spill cost computation provides the cost of load and store instruction (spill code) that is required to spill a live variable. Simplification is a technique for determining the minimum number of registers required by a particular interference graph and the order in which live ranges are assigned registers. It determines if a graph can be assigned with given set of registers. If interference graph is not simplifiable, a live variable with less spill cost is spilled i.e., Load and Store instructions are inserted in the program code. The variables in simplified interference graph are assigned register at the register assignment phase and spill code is inserted for the spilled instructions.

The dead code elimination, peephole optimization, common sub-expression elimination, etc, are done by appropriate modules. Assembly level program is generated by Quad2Asm module. The detail description of original Jackcc compiler and modifications made in the proposed work is given in Appendix B.

The work proposed in this thesis includes the modification of Jackcc compiler to extract fine grain threads and to create schedules for multicore processors. Once compiler successfully identifies parallelization opportunities (fine grain threads) in the program and creates the schedules for multicore processors, a runtime system is required to monitor the execution behavior of parallelized program and fix any miss-speculations that might happen. The runtime speculation engine follow the static parallelization frame work as shown in Figure 9. The parallelized code generated during compile time is later executed along with a runtime speculation engine to monitor execution and roll-back in case of any miss-speculations.

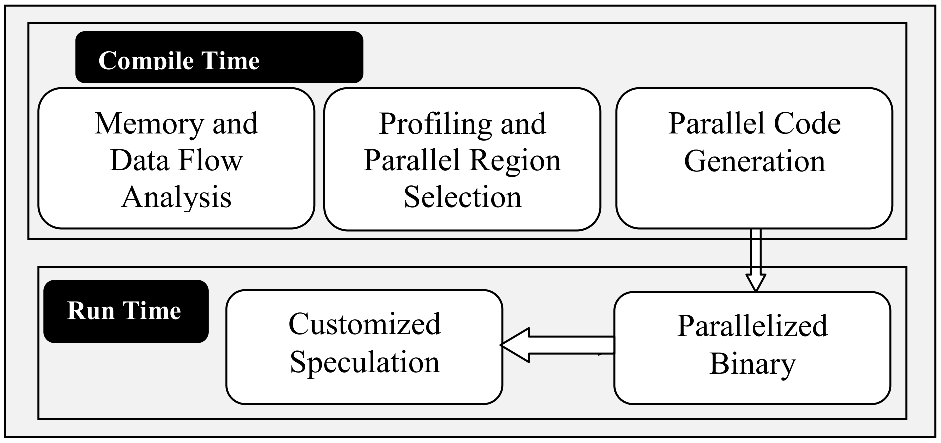


Figure 9. Run Time Support

3.3 Benchmarks

The test cases that are used to evaluate the proposed work are taken from RAW benchmark suite [104][105], and are modified to make it compatible with Jackcc compiler. RAW benchmark suite is designed as part of RAW project in MIT and are maintained under CVS (Concurrent Versions Systems), to facilitate benchmarking and comparing reconfigurable computing systems.

The RAW benchmark suite contains twelve programs designed to facilitate comparing, validating, and improving reconfigurable computing.

TABLE VI. RAW BENCH MARK SUITE

Benchmarks	Used
Bheap	X
Bubble sort	X
DES encryption	√
FFT (Integer fast fourier transform)	√
Graph Problem (ssp-single source shortest path, spm-multiplicative shortest path)	X
Integer Matrix Multiplication	√
Jacobi Relaxation	X
Live (Game based on matrix)	X
Merge Sort	√
NQueens	X

3.4 Experiment Evaluation

The result discussed in this thesis is based on the simulated model of the target architecture, with Dual core and Quad core. The results are shown for four extreme test cases of the benchmark suite such as DES, Integer matrix multiplication, Fast Fourier transform and Merge sort. The results with three active cores on a quad core machine are presented to illustrate the power optimization possibility.

Definition of work and time

Definition 1: The work is defined as unit of task involving execution of finite number of instructions of a given program. A work with n instruction will usually have a mix of computation instructions and data access instructions. We assume that out of n instructions p is number of computational instructions and c is number of data access instructions.

Definition 2: The execution time is defined as the number of CPU cycles spent for completing the work. The total execution time t_n is summation of total computation time t_p and total data access time t_c.

$$t_n = \sum_{p=1}^{p} t_p + \sum_{c=1}^{c} t_c \qquad (01)$$

The execution time is computed by multiplying the number of instructions in the sub-block and average time taken to execute an instruction. Though each instruction execution may take different amount of time, without loss of generality, it is assumed that average time taken to execute an instruction is constant.

Amdahl’s law for multicore architecture proposed by Hill-Marty [11] is used for analyzing the speedup performance. The result is normalized with respect to “Base Core Equivalents (BCE) proposed in the Hill-Marty model. The speedup performance of n-BCE single core processor is 1.

Woo-Lee model [12] is used for checking the energy efficiency of the proposed approach, and performance per power. The model for power consumption with n-cores considers the fraction of power k that a core consume in idle state ($0 \leq k \leq 1$). It is assumed that a core in active state consumes 1 unit of power, i.e., the amount of power consumed by one core during the sequential computation of program is 1 unit, while the remaining $(n - 1)$ cores consume $(n - 1) * k$ *units*. Thus, during the sequential computation phase, the n-core processor consumes 1 +

$(n-1) * k$ *units* of power. In the parallel computation phase, *n core* processor consume *n units* of power. Because it takes $(1-f)$ and f/n to execute the sequential and parallel code, respectively, the formula for average power consumption W in watt is given in equation (02), where f is the fraction of computation that can be parallelized ($0 \leq f \leq 1$).

$$W = \frac{1 + (n-1)k(1-f)}{(1-f) + \frac{f}{n}} \tag{02}$$

The model for performance per watt (*Perf* / *W*), represents the performance achievable at the same cooling capacity and is reciprocal of energy, as the *performance* is reciprocal of execution time. The Perf / W for multicore is given in equation (03). The Perf / W of single core processor is 1.

$$\frac{Perf}{W} = \frac{1}{1 + (n-1)k(1-f)} \tag{03}$$

The communication cost is calculated if the dependent fine grain threads are executed on different cores. The cost of communication depends on the total number of variables shared by the fine grain threads (Nv), total number of times a core communicates with a different core (Ntc), and architecture dependent communication latency (cf). The communication cost is formalized as in equation (04).

$$\text{Communication cost is} \begin{cases} \textbf{Zero:} \text{ if all dependent fine grain threads are scheduled on to same core.} \\ \textbf{Nv*Ntc*cf:} \quad \text{Otherwise} \end{cases} \tag{04}$$

The equations given in Section 3.4 used to show the result in later chapters.

CHAPTER 04 Compiler Assisted Parallelization and Optimization for Multicore Architecture

This chapter briefly explains the techniques to achieve the proposed research objectives excluding the implementation details and results. The implementation details and results are presented in later chapters.

The proposed work introduces two additional phases to the normal flow of compiler as shown in Figure 10. A fine grain thread extractor phase to create parallel regions and the scheduler phase to create schedules for multiple cores. The register allocation phase is modified to view the private register files of individual cores.

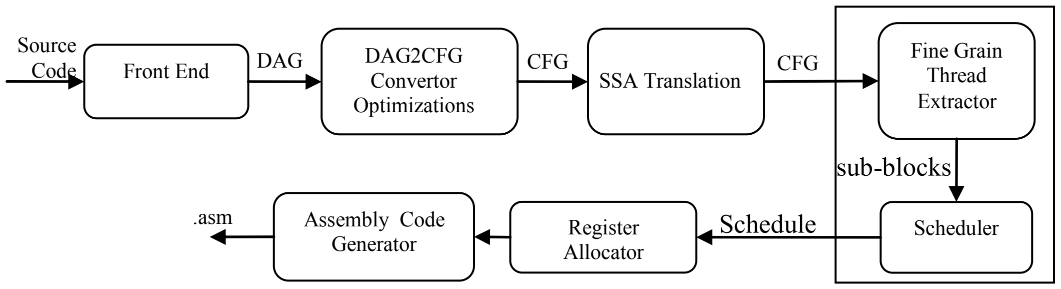


Figure 10. Modified Flow of Compiler to Create Disjoint Sub-blocka and to Create Schedules

Front end of a compiler converts a high level language source code (C Program) to Direct Acyclic Graph (DAG). The DAG is used for creating control flow graph consisting of basic blocks. CFG is utilized for loop unrolling, dead code elimination, common sub expression elimination and generation of SSA form program. The fine grained thread extractor carries out the dependency analysis on SSA form program and creates parallel regions (sub-block) within the basic block. Since the sub-blocks within a basic block can be executed in parallel, are termed as fine grained thread. The fine grained extractor module produces the Sub-block Dependency Graph (SDG) for global scheduling and sub-blocks for local scheduling. The fine grained thread can be scheduled using scheduler. The schedules are utilized for identifying the individual threads for the individual core and for these thread register allocation is carried out by register allocator module. Finally assembly code generator module produces the assembly code.

As indicated in the diagram, the sub-blocks / SDG are created from SSA form program after analyzing the dependencies. Instead of creating SSA form and then carrying out the analysis to form the sub-blocks / SDG which requires two separate passes to perform the operation, SSA translation and Fine grained extraction module is merged where the sub blocks are formed at the time of SSA form generation resulting into compilation time reductions.

Since the sub-blocks are disjoint, when scheduled on different cores do not communicate with each other resulting into reduced communication latency and preventing race condition [03][05].

In order to reduce the power consumption, an approach to find the optimal number of cores required for execution of a program without compromising with the speed-up [02][05] has been proposed.

4.1 Fine Grain Thread Extractor

The fine grain thread extractor module acts upon the basic blocks (B_p) of control flow graph (CFG). The instructions in each basic block of CFG are analyzed for dependency to create disjoint sub-blocks. In Figure 11, the CFG has four basic blocks B_1, B_2, B_3 and B_4. The disjoint set operations are applied on each basic block to form sub-blocks SB_i. The sub-block SB_i belonging to basic block B_p is refereed as SB_iB_p.

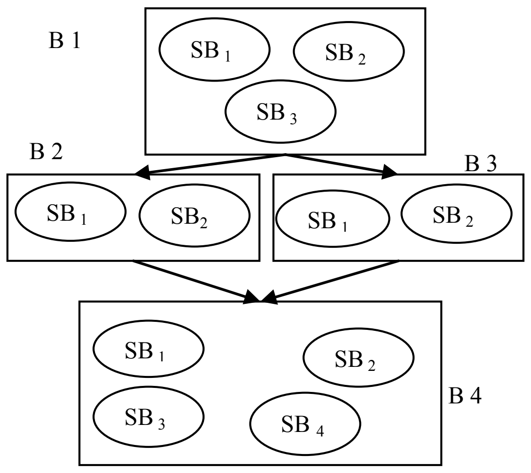
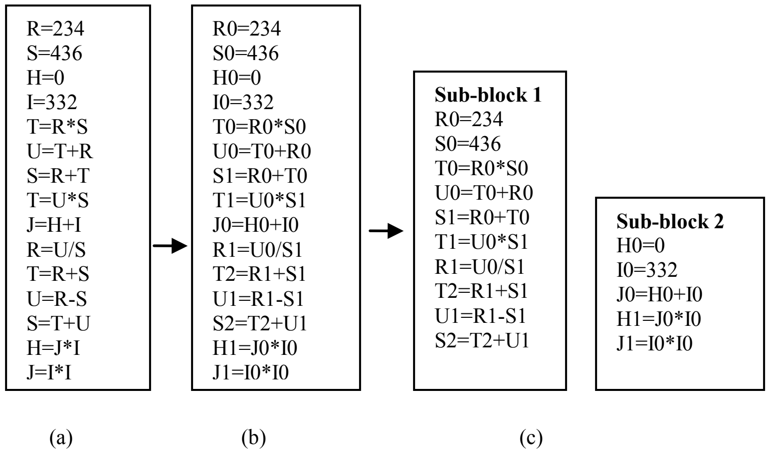


Figure 11. Control Flow Graph with of Basic Block B_p Sub-blocks SB_i

In general programs have three kinds of data dependence [6]: true dependencies (Read-after-Write), anti-dependencies (Write-after-Read), and output dependencies (Write-after-Write). Static single assignment (SSA) is an intermediate representation of the program, wherein each variable has only one definition in the program text. This is an improvement on the idea of def-use chains. The advantages of SSA form program is that it removes output dependency and anti dependency between instructions. Compiler only needs to look for true dependencies while extracting parallelism. Figure 12.a gives a program segment which is converted to SSA form program and is shown in Figure 12.b. The SSA form program is analyzed and disjoint set operations are applied to produce two sub blocks which are shown in Figure 12.c.



(a)

```
R=234
S=436
H=0
I=332
T=R*S
U=T+R
S=R+T
T=U*S
J=H+I
R=U/S
T=R+S
U=R-S
S=T+U
H=J*I
J=I*I
```

(b)

```
R0=234
S0=436
H0=0
I0=332
T0=R0*S0
U0=T0+R0
S1=R0+T0
T1=U0*S1
J0=H0+I0
R1=U0/S1
T2=R1+S1
U1=R1-S1
S2=T2+U1
H1=J0*I0
J1=I0*I0
```

(c)

```
Sub-block 1
R0=234
S0=436
T0=R0*S0
U0=T0+R0
S1=R0+T0
T1=U0*S1
R1=U0/S1
T2=R1+S1
U1=R1-S1
S2=T2+U1
```

```
Sub-block 2
H0=0
I0=332
J0=H0+I0
H1=J0*I0
J1=I0*I0
```

Figure 12. (a) Non SSA Program (b) SSA Form Program (c) Disjoint sub-blocks

Definition 3: Disjoint sub-block

Let S be a list of disjoint sub-blocks. That is, for all sub-blocks $\{SB_i : i \in \mathbf{I}\}$ in S indexed by I, the intersection of these sub-blocks is empty: $SB_i \cap SB_j = \emptyset$. The union of all sub-blocks in list is given by,

$$\cup \mathrm{SBi} = \cup \{(\mathrm{x}, \mathrm{i}) : x \in SBi\} \qquad (05)$$

Where i serves as an auxiliary index that indicates which SB_i the instruction x came from.

Creating sub-blocks has some positive offshoots.

- This conforms to the principal of spatial locality as the closely related or dependent instructions are grouped together in sub-blocks.
- Minimizes the cache coherence problems as the instruction stream of a sub-block scheduled to a core is not dependent on what is scheduled on the other cores at a time.
- This in turn reduces the need for communication among the cores.
- Since for the instruction stream, dependency analysis is done in the compilation phase the hardware level reordering overhead is reduced. This makes the technique power aware.

The sub-blocks are disjoint within a basic block of CFG, but the sub-blocks across the basic blocks need not be disjoint. The non-disjoint sub-blocks should be executed in order. Inter block scheduling is a process of finding the non-disjoint sub-blocks across the basic blocks and define AFTER relation between them.

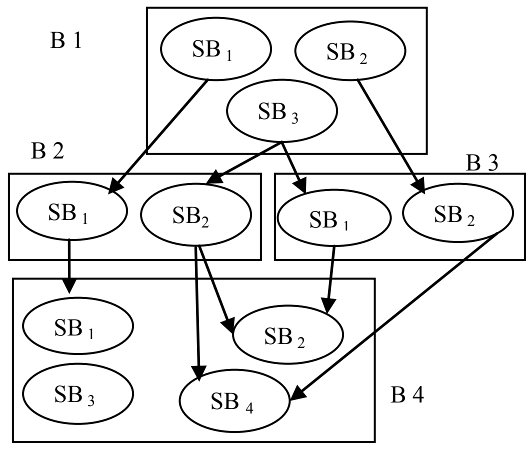
Definition 4: AFTER

The sub-block SB_j of block B_q should be scheduled after SB_i of basic block B_p where $p \neq q$, if one or more instructions in SB_j has true dependency with the instruction in SB_i. The SB_j is AFTER related with SB_i and it is denoted as $SB_i \rightarrow SB_j$.

A sub-block dependency graph (SDG) is constructed to guide the inter block scheduler.

Definition 5: Sub-block Dependency Graph (SDG)

The SDG is graph G (V, E), where vertex $v_i \in V$ is a sub-block SB_iB_p, and the directed edge $e \in E$, is drawn between vertex $SB_i \in B_p$ and vertex $SB_j \in B_q$ where $p \neq q$. We say SB_jB_q is AFTER related to SB_iB_p. SDG is represented as dependency matrix. In dependency matrix all sub-blocks are arranged in first column and rest of the columns entry indicates dependency list. If the sub-block SB_jB_q is dependent on sub-block SB_iB_p, then SB_iB_p is added in the dependency list of SB_jB_q, meaning SB_jB_q should be scheduled only after SB_iB_p completes its execution. The sub-block SB_jB_q can be scheduled only if the list is empty otherwise it should wait till the list becomes empty.



(a)

	Sub-blocks	Dependency List	
1	SB_1B_1		
2	SB_2B_1		
3	SB_3B_1		
4	SB_1B_2	SB_1B_1	
5	SB_2B_2	SB_3B_1	
6	SB_1B_3	SB_3B_1	
7	SB_2B_3	SB_2B_1	
8	SB_1B_4	SB_1B_2	
9	SB_2B_4	SB_1B_3	SB_2B_2
10	SB_3B_4		
11	SB_4B_4	SB_2B_2	SB_2B_3

(b)

Figure 13. (a) Sub-block Dependency Graph (b) Depenedency Matrix

4.2 Scheduling

In the proposed compiler framework, scheduling phase follows the SSA translation and sub-block creation phases. The scheduler generates schedule for each core. Each schedule consists of list of sub-blocks that can be scheduled on a core. Table VII shows, generation of two schedules for a dual core machine. Scheduling sub-block may be local or global. The local scheduling is termed as Intra block scheduling, where only sub-blocks inside a basic block are considered for scheduling. Global scheduler (Inter block scheduler) identifies all the independent sub-blocks in a CFG and formulates the schedule. Four novel inter block schedulers are proposed in the current research. The Height Instruction Count Based scheduler (HIB), Dependent Sub-block First based scheduler (DBS), Most Dependent Sub-block First scheduler (MDS) and Largest Latency Sub-block First scheduler (LLSF).

TABLE VII. LIST OF SUB-BLOCKS GENERATED BY MDS SCHEDULER

Dual Core		3 Active Cores			Quad Core			
Core 1	Core 2	Core 1	Core 2	Core 3	Core 1	Core 2	Core 3	Core 4
SB_3B_1 SB_1B_1 SB_1B_3 SB_2B_3 SB_1B_4 SB_2B_4	SB_2B_1 SB_2B_2 SB_1B_2 SB_3B_4 SB_4B_4	SB_1B_1 SB_2B_3 SB_1B_2 SB_1B_4	SB_2B_1 SB_2B_2 SB_3B_4 SB_2B_4	SB_3B_1 SB_1B_3 SB_4B_4	SB_1B_1 SB_2B_3 SB_1B_4	SB_2B_1 SB_2B_2 SB_2B_4	SB_3B_1 SB_1B_3 SB_4B_4	SB_3B_4 SB_1B_2

In general the global scheduler selects the sub-block i of basic block Bp (SB_iB_p) from the sub-block dependency matrix if its dependency list is empty. Sub-block dependency matrix is jagged matrix representation of SDG as shown in Figure 13.b. Once SB_iB_p is scheduled and completes its execution, its corresponding entries are removed from dependency list.

The decision of scheduling a sub-block on a core is based on the invariants such as scheduling latency, computed ready time (TRdy) & finish time (TFns) of the sub-block SB_iB_p. The schedule time (TSch) of sub-block and total scheduled time of core (TSct) are also taken into consideration to check the availability of a core to schedule the sub-blocks. Height and schedule latency of sub-blocks are computed in bottom-up fashion. The total scheduled time of core (TSct) is the time taken by a core to complete the execution of the sub-blocks currently scheduled on it. TSct is computed in top-down fashion on SDG. TSct suggests the time at which next sub-block could be scheduled on to the core. The ready sub-block is scheduled on a core with lower TSct.

The height of the sub-block SB_iB_p is one more than maximum height of all its immediate successors (AIS).

$$Height_i = Maximum(Height(AIS)) + 1 \quad (06)$$

The equation (07) gives the predicted finish time (TFns) of a sub-block SB_iB_p.

$$TFns_i = C_i + TSch_i \quad (07)$$

Where C_i is total cycle time of ith sub-block and $TSch_i$ is schedule time of ith sub- block.

The ready time (TRdy) of a sub-block SB_iB_p is given below in equation (08). Ready time of a sub-block is the time at which sub-block is free from all its dependencies and ready to be scheduled on a core. i.e. maximum finish time of all its immediate predecessors (AIP).

$$TRdy_i = Maximum(TFns(AIP)) \quad (08)$$

The schedule latency (L_i) of a sub-block is given in equation (09). The schedule latency of leaf sub-block in SDG is, total number of instruction inside the leaf sub-block. The schedule latency of SB_iB_p is sum of maximum latency of all its immediate successors (AIS) and total number of instructions inside the sub-block SB_iB_p or total cycle time of ith sub-block SB_iB_p.

$$L_i = Maximum\ (L(AIS)) + C_i \quad (09)$$

The total scheduled time of a core *k* (TSct) is given in equation (10).

$$TSct_k = TSct_{k-1} + \quad (10)$$

Where $TSct_{k-1}$ is current schedule time of the core and C_i is total cycle time required by SB_iB_p.

4.2.1 General Criteria To Create Schedule

Case 1: All sub-blocks are dependent

If all the sub-blocks are dependent then they need to be scheduled on the same core which result in poor utilization of multicore environment.

We follow global scheduling heuristics such as Dependent Sub-block Based (DSB), Most Dependent Sub-block First (MDS), Height and Instruction Count Based (HIB) and Lowest Latency First (LLFS), scheduling heuristics to schedule the sub-blocks on the core.

By following certain heuristics like DBS, MDS, LLSF and HIB optimal execution time can be achieved.

Case 2: All sub-block are independent

The sub-blocks inside the basic blocks are independent and can be scheduled in parallel. In this case schedule is created taking number of cores into consideration. Here, the sub-blocks are merged such that resulting execution time is low.

It should be noted that all sub-blocks across the basic blocks are not independent.

Case 3: Not all sub-blocks are dependent

i. *Schedule all dependent sub-blocks on to same core*

Facilitate less communication latency but results in imbalance usage of core.

ii. *Schedule heuristically*

We follow global scheduling heuristics such as Dependent Sub-block Based (DSB), Most Dependent Sub-block First (MDS), Height and Instruction Count Based (HIB) and Lowest Latency First (LLFS), scheduling heuristics to schedule the sub-blocks on the core.

Case 4: Instruction in the sub-block depend on branch instruction.

i. *Predict branch taken or not taken and schedule accordingly*

Predicting a branch taken or not taken is difficult at compile time. Few compiler prediction technique are proposed in history but they have proved not efficient.

ii. Schedule the sub-blocks in both path of branch instruction (branch taken or not taken) on different cores without worrying the branch result and let the runtime environment decide whether to schedule the sub-block or not.

iii. Keep option to schedule the sub-blocks of the branch instruction on same cores and let the runtime environment decide either of them based on result of branch.

Option ii and iii need runtime support and we follow second option.

4.2.2 Sequential Program Execution and Its Analysis

This section, presents the results of executing sequential program compiled using original Jackcc compiler on n-BCE multicore processor. The relative results in terms of speed-up, power consumption and performance per power achieved by executing sequential program on one of the core of dual core and quad core processor is shown in the Figure 14 and Figure 15.

The equations used for computing the results are explained in Section 3.4. The result shown is relative to n-BCE single core processor. The speed-up performance ($Perf_s(r)$) of n-BCE single core processor is 1. The amount of power consumed by n-BCE single core processor during the sequential computation of program is 1 unit. The performance per watt (Perf /W) of single core processor is 1. If the program is compiled for single core and executed on n-BCE multicore processor (either dual or quad core processor) the speed-up ($Perf_m(r)$) decreases. The speed-up will be ($0 \leq Perf_m(r) < 1$), this is because, a core in multicore processor is less powerful than n-BCE single core processor. This results in increased execution time and power consumption as shown in Table I. The performance per watt also decreases and will be less than 1 compared to n-BCE single core processor.

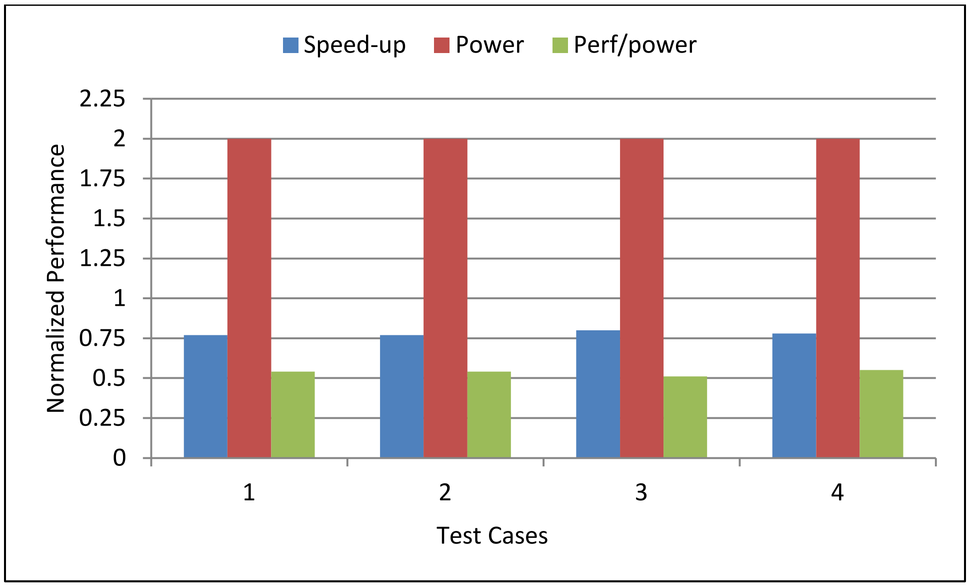


Figure 14. Results of Testcases Compiled by Original Jackcc for Dual Core Processor

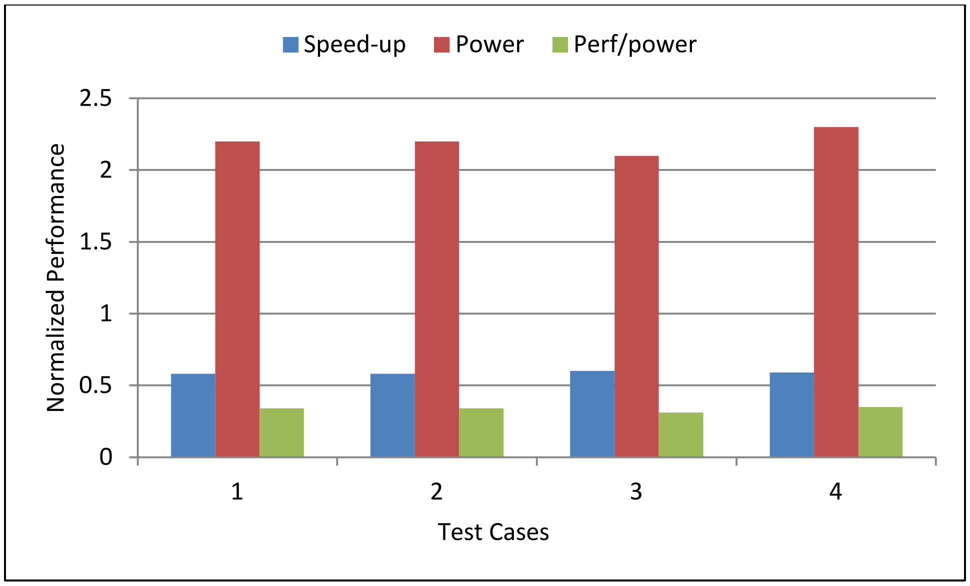


Figure 15. Results of Testcases Compiled by Original Jackcc for Quad Core Processor

4.2.3 Power Optimization

The output of the proposed schedulers is a list of sub-blocks (schedules) to be scheduled on the cores. For a N core machine the scheduler generates M schedules, where $M \leq N$. The Intra block scheduler and Inter block schedulers are designed to perform power optimization. It tries to find the optimal number of cores on which sub-blocks can be scheduled thus resulting in reduced power consumption. For example, if the speed-up achieved using N cores can be achieved using N-1 cores, then only N-1 cores of the N core machine are used to execute the given program, either by keeping the N^{th} core idle or utilizing it for some other computation. This is true for any N number of cores where N= 2,4,8,16....

The Intra block scheduler and Inter block scheduler follow the strategy of merging the sub-blocks scheduled on different cores to reduce the usage of number of cores without compromising much with the speed-up. This is possible when the total execution (T_{i+m}) of one core is $T_{i+m} \geq \sum_{n=i=0}^{\infty}(T_{i+n})$ where $T_i, T_{i+1}, T_{i+2}, T_{i+3.....}$ T_n are execution time of n cores and $i+m \neq i+n$. As sub-blocks in different schedules may not be necessarily disjoint, the scheduler cannot merge the sub-blocks in different schedules randomly. Instead it heuristically distribute the sub-blocks onto different schedules by keeping total execution time of cores equal.

The scheduler generate schedules iteratively starting from 2 core to N core. If the total execution time T_i of i^{th} core is same as total execution time of T_{i-1} $(i-1)^{th}$ core for $0<i\leq N$ the scheduler stops creating schedules for other cores.

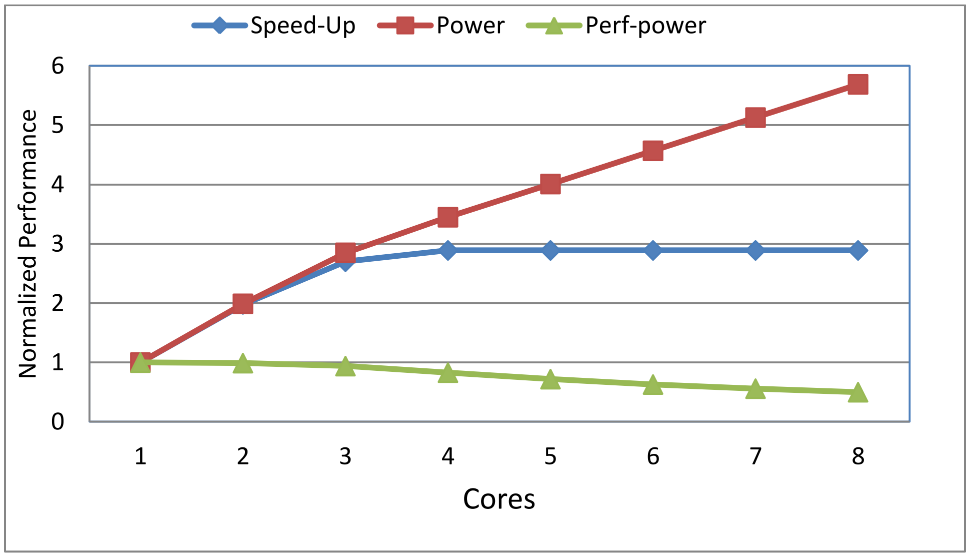


Figure 16. Power Optimization

The graph given in Figure 16 shows the speed-up achieved by scheduling a sequential program on 8 core processor. The gain in speed-up will reach its threshold on four cores. Performance per power will start declining as power consumption increases when all 8 available cores are utilized. The proposed schedulers will stop creating schedules for other cores when threshold in speed-up is reached. i.e., the scheduler will create 4 schedules. For power critical architectures the compiler can be tuned to stop creating schedules for three core alone. This is done at the cost of speed-up.

4.3 Register Allocation

The fine grained threads that are scheduled on to different cores need to be allocated registers from respective register file of the core on which they are scheduled. It is proposed to develop register allocation strategy for fine grained threads which can be scheduled on multicore processor. Four different register allocation heuristics for multicore processor architecture is explained in this section.

The first heuristic uses schedule generated by the scheduler for register allocation. Instructions in each sub-block are allocated locally using Chaitin's approach and are scheduled as directed by the scheduler. This approach leads to reduced compilation time but execution time is increased as individual sub-blocks are to be assigned thread requiring data movement.

The second heuristics integrates register allocation with global scheduling [34]. The goal is to eliminate the phase ordering problem and to overcome limitations which lead to poor optimizations. The scheduler allocates the sub-blocks in the dependency graph to multiple cores selectively, taking register requirement, dependencies and the order of execution into account.

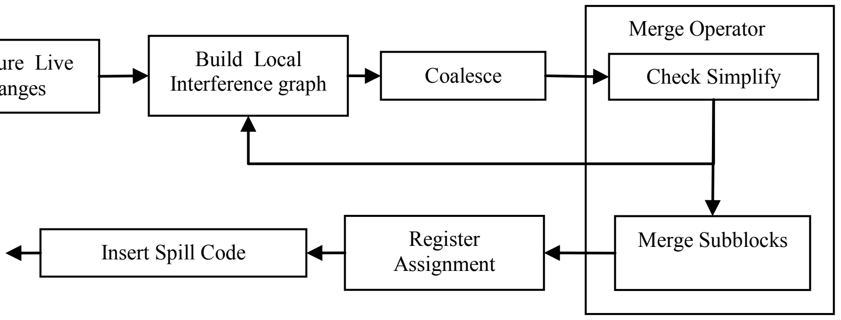


Figure 17. Modified Register Allocation Framework

Heuristic 3 and heuristic 4, follow the proposed register allocation framework as shown in Figure 17. To facilitate the global scheduling, these heuristics incrementally merges the sub-blocks in the sub-block list generated by scheduler to produce a hyper sub-block list H (h1,h2,h3....hx). The global interference graph is constructed incrementally by merging individual local interference sub-graphs one by one and checking if the resulting sub-graph is k-colorable. The algorithm incrementally merges the sub-blocks to form hyper sub-blocks using merge operator. Since the Hyper sub-blocks are scheduled on a single core, it ensures temporal locality and reduces memory reference. The Hyper sub-blocks are k-colorable which cause zero spilling and instructions remain inside private memory of individual cores till all the instruction commits without doing external memory reference. The variables in hyper sub-block are assigned register at the register assignment phase. These heuristics help to produce the optimized code at cost of increased compilation time.

In heuristic 3 the hyper sub-blocks are created by merging the sub-blocks. Merging of sub-block is carried out by coalescing the interference graph of the sub-blocks and by checking the sub-block dependency, Ready_time (TRdy) and Finish_time (TFns) of the sub-blocks that are being merged. In this heuristics, similar to Chaitin's approach, initially a global interference graph (hyper sub-block) is built and then the interference graph is simplified to make it k-colorable. If the interference graph is not k-colorable, it is simplified and spill code is inserted. Since each hyper sub-block can be assigned a thread, it improves the execution time as compared to approach 1 but may add more spill code as interference graph may not be k-colorable. The proposed fourth heuristic overcome the limitations of heuristic 1 & 3. In this approach the hyper sub-blocks are created by adding simplifiability condition to the heuristic 3. The simplifiability condition ensures that the interference graph of the hyper sub-blocks are k-colorable resulting in zero spill code.

An example to illustrate the proposed register allocation approach for the cores having four registers is given Figure 18. The schedule created by the global scheduler for dual core machine is shown in Table VIII (a). The hyper sub-block generated by using the schedule list is shown in Table VIII(b). Table IX exposes the instructions in sub-blocks SB_2B_1 and SB_2B_3 scheduled on core 2 of dual core machine.   The 3-colorable interference graphs for sub-blocks SB_2B_1 and SB_2B_3 is shown in Figure 18.a and Figure 18 b respectively. The 3-colorable interference graph of merged sub-blocks is shown in Figure 18 c.

TABLE VIII. (A)SUB-BLOCK LIST GENERATED FOR DUAL CORE PROCESSOR (B) HYPER SUB-BLOCKS WHOSE INTERFERENCE GRAPH IS K-COLORABLE

Dual Core	
Core1	Core2
SB_3B_1	SB_2B_1
SB_1B_1	SB_2B_3
SB_1B2	SB_2B_2
SB_1B_3	SB_4B_4
SB_1B_4	SB_3B_4
SB_2B_4	

(a)

Dual Core	
Core1	Core2
SB_3B_1 SB_1B_1	SB_2B_1 SB_2B_3
SB_1B_2 SB_1B_3 SB_1B_4	SB_2B_2 SB_4B_4 SB_3B_4
SB_2B_4	

(b)

TABLE IX. SUB-BLOCKS

SUB-BLOCK SB_2B_1	SUB-BLOCK SB_2B_3
F0=11;	U0=G1+F1;
G0=42;	V0=G1+H1;
H0=F0/G0;	W0=U0+V0;
I0=G0+H0;	V1=W0*V0;
G1=G0-I0;	U1=W0-V1;
H1=G1+I0;	W1=U1*W0;
F1=G1+H1;	

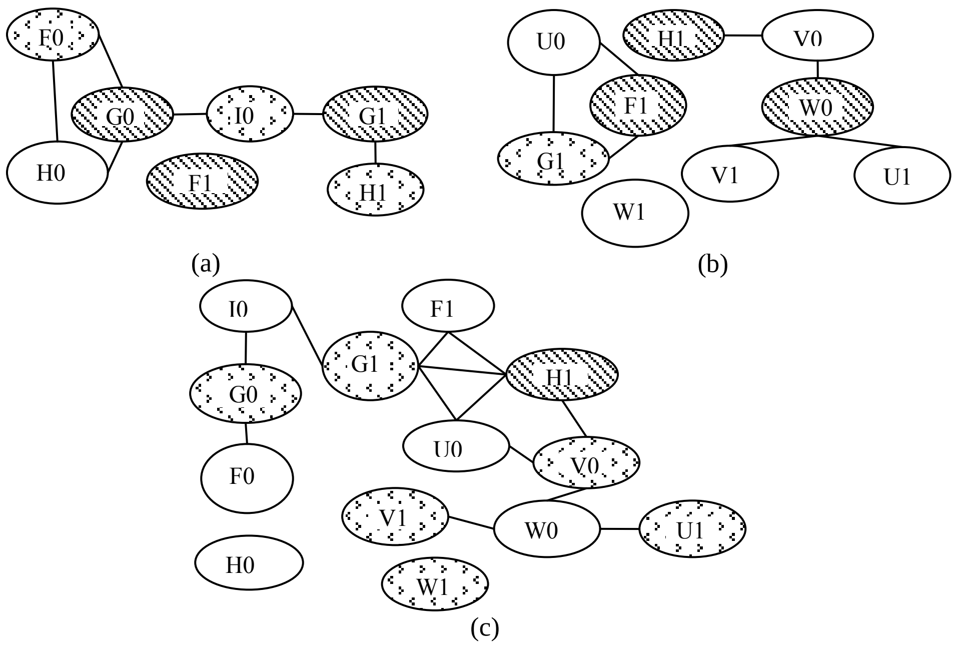


Figure 18. (a) Interference Graph of Sub-block SB_2B_1 (b) Interference Graph of Sub-block SB_2B3 (c) Interference Graph of merged Sub-blocks

The details of the steps involved in implementing scheduling and register allocation techniques are explained in chapters 5, 6,7, and 8.

CHAPTER 05 Fine Grain Thread Extractor

The implementation details of fine grain thread extractor module is discussed in this chapter. The Figure 10 in chapter 4 depict the steps involved in this process. The fine grain thread extractor module creates the disjoint sub-blocks and sub-block dependency graph (SDG). The sub-block dependency graph (SDG) is used for global scheduling (Inter block Scheduling).

5.1 Creating Disjoint Sub-blocks

Two approaches to create the disjoint sub-blocks or fine grain threads has been proposed and is briefly discussed in following sections.

i. In the first approach, a SSA form program is taken as input. A separate pass is used for dependency analysis and sub-blocks creation.
ii. In the second approach, dependency analysis and sub-block creation is done along with variable renaming step in the generation of SSA form program. This approach has benefit in terms of compilation time as it avoids the need for an extra pass.

5.1.1 Approach 1 To Create Disjoint Sub-blocks

The operations for creating disjoint sub-blocks is performed after SSA translation phase of the compiler. In this approach, disjoint set operation is performed on instructions which are represented using quad data structure belonging to basic block also known as *Arena* to create a list S of sub-blocks. The list S is given to scheduler to create schedules for cores of a multicore processor.

The disjoint set operations such as makeSet, Union, and groupQuads are applied on instructions in each basic block B_p. Each sub-blocks are identified by one of the designated instruction, which is called as representative instruction of the sub block. An example to depict the steps involved in sub-blocks creation is shown in Figure 19.

Let the basic block B_p contains n instructions i.e. x_1, x_2. ..x_n.

- The makeSet operates upon a basic block B_p containing n instructions and divides it into n sub-blocks with each sub-block containing one instruction. These sub-blocks are represented in form of a list S_p {SB_1B_P, SB_2B_P, SB_3B_P.... SB_nB_P} containing n sub blocks. Since the sub-blocks are disjoint, it is required that instruction x_i is present in only one sub block say SB_iB_P. The instruction x_i is designated as representative of the sub-blocks SB_iB_P. The output of makeSet module when applied on basic block with 15 instructions is shown in Figure 19.

- The groupQuads module operates on the list S_p with n instructions. This module checks for true dependency between the instructions and uses Union operation to create m disjoint sub-blocks, where $m \leq n$.
- The Union operation merges the dependent sub-blocks to form m number of disjoint sub-blocks. If two dynamic sets (sub-blocks) SB_iB_P and SB_jB_P contains the instruction x_i and x_j, the two sub-blocks are assumed to be disjoint prior to the Union operation and hence their set representatives are different. The union operation is performed by changing the representative of the sub-block, i.e. one representative is selected from either of the sub-blocks as shown in Figure 19.

Algorithm1: To create initial list of sub-blocks with one quad in each sub-block.

```
makeSet(Arena A,List S)
 begin
  for each quad in A
  repeat
       Create N sub-block making itself as representative to form the list S;
  End for
  call groupQuads(List S);
End
```

Algorithm2: To group the quads based on true dependency

```
groupQuads(List S, number of quads)
begin
   for each sub-block i =0 to number of  quads in the list S repeat
   for each sub-block j=i+1 to number of quads in the list s
repeat
   if there is true dependency between i^th & j^th instruction
then
   if the sub-block representative are not same
then
          call      Union(S, i, j);
      return S;
  End for
  End for
End
```

Algorithm3: To perform disjoint union operation

```
Union(List S,index i,index j)
begin
    copy the j^th instruction to the sub-block to which i^th instruction belong to;
    change the j^th sub-block_rep to i^th instructions sub-block_rep;
 End
```

Instruction	Representative
R0=234	1
S0=436	2
H0=0	3
I0=332	4
T0=R0*S0	5
U0=T0+R0	6
S1=R0+T0	7
T1=U0*S1	8
J0=H0+I0	9
R1=U0/S1	10
T2=R1+S1	11
U1=R1-S1	12
S2=T2+U1	13
H1=J0*I0	14
J1=I0*I0	15

a. After Makeset

Instruction	Representative
R0=234	1
S0=436	2
H0=0	3
I0=332	4
T0=R0*S0	1
U0=T0+R0	1
S1=R0+T0	1
T1=U0*S1	8
J0=H0+I0	9
R1=U0/S1	10
T2=R1+S1	11
U1=R1-S1	12
S2=T2+U1	13
H1=J0*I0	14
J1=I0*I0	15

b. 1st Iteration of union and groupQuad

Instruction	Representative
R0=234	1
S0=436	1
H0=0	3
I0=332	4
T0=R0*S0	1
U0=T0+R0	1
S1=R0+T0	1
T1=U0*S1	1
J0=H0+I0	9
R1=U0/S1	11
T2=R1+S1	1
U1=R1-S1	1
S2=T2+U1	1
H1=J0*I0	14
J1=I0*I0	15

c. 2nd Iteration of union and groupQuad

Instruction	Representative
R0=234	1
S0=436	1
H0=0	3
I0=332	3
T0=R0*S0	1
U0=T0+R0	1
S1=R0+T0	1
T1=U0*S1	1
J0=H0+I0	3
R1=U0/S1	1
T2=R1+S1	1
U1=R1-S1	1
S2=T2+U1	1
H1=J0*I0	3
J1=I0*I0	3

d. 3rd Iteration of union and groupQuad

Figure 19. Steps in Creation of Disjoint Sub-blocks

5.1.2 Approach 2 To Create Disjoint Sub-blocks

The approach presented in this section creates the disjoint sub-blocks while translating non SSA program to SSA form program eliminating extra pass requirement of Approach 1 and thus resulting in reduced compilation time. The modified work flow of compiler for this approach is shown in Figure 20.

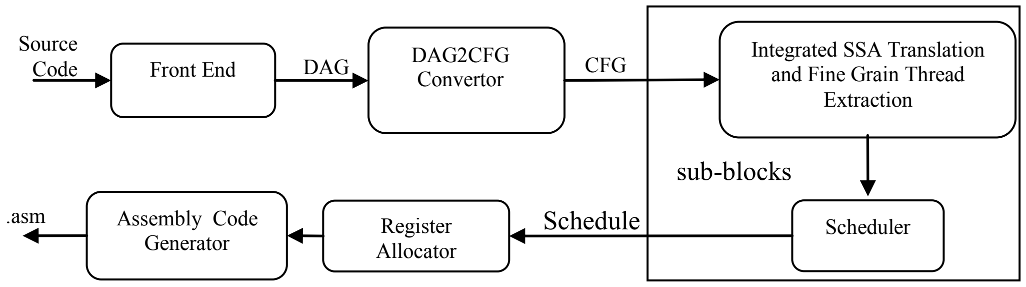


Figure 20. Integrated SSA Translation and Fine Grain Thread Extraction

The modified version of *variable renaming* algorithm of SSA translation pass is presented. The proposed algorithm performs variable renaming and sub-blocks creation tasks simultaneously during the SSA translation pass. The first task is accomplished by traversing each basic block of the CFG which has Phi (ϕ) functions inserted by Phi_function( ) module and rename each variable in a way that each use corresponds to exactly one definition. Each definition is renamed with a new version of that variable. Second job is accomplished by updating the use of the renamed variable, which in turn helps in identifying the instructions having true dependency (RAW). These dependent instructions are collected to form disjoint sub- blocks.

Algorithm 4: Integrated variable renaming and sub-block creation

```
Start from first statement in the block
 Loop until all statements are parsed
 Scan current statement in the block
 Rename the definition and use in current statement
Now check in current statement, variables on right hand side
    if(number of variables on right hand side=0)
       Create new sub-block(SBBn) // nth sub-block
       SBBn[0]=current statement number
       Number of statements in current sub-block +=1;
    End if
    else if(number of variables on right hand side y0 ==1)
            if(y0 is defined in the current block)
               find(sub-block in which statement defining y0 is residing)
                 //Suppose this returns SBBy
     SBBy[Number of statements in sub-block SBBy++]=current statement number
            End if
            else
               Create new sub-block(SBBn) // nth sub-block
               SBBn[0]=current statement number
               Number of statements in current sub-block +=1;
            End else
     End else if
     else if(number of variables on right hand side==2)
          if(y0 and z0 are defined in the current block)
             SBBy=find(sub-block in which statement defining y0 is residing)
             SBBz=find(sub-block in which statement defining z0 is residing)
            if(SBBy==SBBz)  //Both are defined in same block
     SBBy[Number of statements in sub-block SBBy++]=current statement number
            Endif
            else            //Both are defined in different block
              Copy all statements in SBBz to SBBy
              Number of statements in SBBy=No of statements in SBBy + No of
                    statements in SBBz
     SBBy[Number of statements in sub-block SBBy++]=current statement number
             Delete SBBz
          End else
        End if
        else if(only y0(or z0) is defined in current block)
             find(sub-block in which statement defining y0 is residing)
             //Suppose this returns SBBy
     SBBy[Number of statements in sub-block SBBy++]=current statement number
       End elseif
       else  //In this case neither of y0 or z0 is defined in current block
           Create new sub-block(SBBn)   //n stands for nth sub-block
           SBBn[0]=current statement number
           Number of statements in current sub-block +=1;
      End else
End
```

The sub-blocks created for four extreme test cases of the benchmark suite such as DES, Integer matrix multiplication, Fast Fourier transform and Merge sort are given in Table X.

TABLE X. NUMBER OF BASIC BLOCKS AND SUB-BLOCKS FOR THE TEST CASES

Test Cases	Number of Basic Blocks	Sub-blocks					
Test Case 1	6	B1= 3	B2= 3	B3= 2	B4= 3	B5= 2	B6= 6
Test Case 2	5	B1= 4	B2= 3	B3= 3	B4= 4	B5= 3	
Test Case 3	4	B1= 3	B2= 3	B3= 3	B4= 3		
Test Case 4	6	B1= 2	B2= 2	B3= 2	B4= 3	B5= 2	B6=2

5.2 Sub-block Dependency Graph (SDG)

The global scheduling or *Inter block scheduling* is a technique for creating schedules for multiple cores by considering the sub-blocks across the CFG. A sub-block dependency graph (SDG) is constructed to facilitate the global scheduling.
Two approaches are proposed to construct the SDG.

i. In the first approach the sub-block dependency graph is created using the sub-blocks. The approach to create sub-blocks has been discussed in Section 5.1. This approach requires an extra pass for creating SDG.
ii. In the second approach, dependency analysis, sub-block creation and SDG creation is done along with variable renaming of SSA form program translation step. This approach eliminates the need for an extra pass as required in the first approach.

5.2.1 Approach 1 To Create SDG

The basic block is split into sub-blocks. The array *subbl_phi[]* stores the sub-block number to which each Φ function belongs. The dependencies of each sub-block is computed and are stored in *dependency_Block* and *dependency_SubBlock* – while former stores pointers to basic blocks to which a sub-block is dependent upon, latter stores the corresponding sub-block number. The *depcount* stores the number of dependencies. *subBlockGraph[]* is an array of nodes of sub-block dependency graph – one node corresponding to each sub-block. The sub-block (node) in Sub-block Dependence Graph has list of quads, Sub-block number, size, list of children, list of parents, Φ functions belonging to the sub-block and a pointer to the basic block of CFG to which the sub-block belongs.

Constructing sub-block dependence graph involves two steps. The first step involves identification of AFTER dependency between sub-blocks . In second step SDG is created .

Step1:– The basic blocks of CFG are used for computing the dependency. The function *computeDependency()* takes a basic block B_p as input. Each basic block B_p can have number of sub blocks represented in the form of a list $S_p\{SB_1, SB_2, SB_3.... SB_n\}$. For each sub-block SB_i in B_p it will mark all sub-blocks SB_j belonging to any other basic block Bq on which it is AFTER dependent. The Use and Def of each basic block is used in this analysis. If sub-block SB_iB_p contains Φ functions, the Use list of that sub-block is updated with the variables v_i in that Φ(v1,v2...) function. For each Use of the sub-block, the sub-block SB_j and basic block B_q in which that variable is defined is found. It then adds the sub-block SB_iB_p to the list of dependencies of SB_jB_q. It repeats the process for each quad in the sub-block SB_iB_p, computing Use and finding where it was defined. The *def* list is also updated by examining each instruction's definition. The function *addToDependencyList()* will take a sub-block SB_iB_p and the sub-block SB_jB_q on which it is dependent upon and update *dependency_Bloc* and *dependency_SubBlock*.

Algorithm 5: Compute Dependency

```
ComputeDependency()
begin
    for each subBlock SB_i in SetList Sp
    begin
        for each Ø function belonging to sub-block SB_i
        begin
            for each column of  Ø  function, j=1 to num_parents
            begin
                add Ø[j] to the list use[SB_i]
            end for
       end for
       for each quad Q in sub-block SB_i
       begin
           add src_a and src_b of Q to the list  use[SB_i]
       end for
       for each variable v in  use[SB_i]
       begin
           S_p = find sub-block in which v is defined
           addToDependencyList(SBiBp,SBjBq)
       end for
End
```

Step 2: SDG is created by calling a function createSDG() which takes basic block B_p and sub-block list S_p as input. Based on dependency computation of the sub-blocks, it creates nodes for each sub-block SB_i in B_p by adding an edge to the sub-blocks on which it is AFTER dependent making it its parents. The dependency edges are created between the sub-blocks SB_iB_p and those sub-blocks SB_jB_q by adding SB_iB_p to the dependency list of

SB_jB_q in dependency matrix. The Figure 13 a in Chapter 4 depicts an example SDG and its corresponding dependency matrix.

Algorithm 6: Create SDG

```
createSDG( )
begin
    for each subBlockSB$_i$ in SetList Sp
    begin
        Create a node of SDG, say v corresponding to SB$_i$
        for j=1 to depcount[SB$_j$]
        begin
            u=SDG node corresponding to dependency$_{block}$[SB$_i$][j] and sub-block
              dependency$_{block}$[SB$_i$][j]
            add the edge u→v
            include v in list of u’s children
            include u in the list of v’s parents
        end for
    end for
End
```

5.2.2 Approach 2 To Create SDG

The technique discussed to create SDG in this section is an extension of Approach 2 of creating sub-block along with SSA renaming module described in Section 5.1.2. This approach is compiler efficient, because it is performed during SSA translation phase itself as shown in Figure 21.

Algorithm 7: Creating Sub-block Dependency Graph

```
Scan the current instruction or quad
if(variables on right hand side==0)
       if statement has phi function
              for each predecessor of this block do
                     find_sub-block_and_block
                     add_sub-block_to_dependency_list
              endfor
       endif
endif
else if(number of variables on right hand side==1)
       find where this variable is defined
       if(variable is not defined in the current block)
           find_sub-block_and_block
              add_sub-block_to_dependency_list
    endif
endelseif
else if(number of variables on right hand side==2)
       find where these variables are defined
       if(one variable is not defined in current block)
              find_sub-block_and_block
              add_sub-block_to_dependency_list
       endif
       else if(no variable is defined here)
              find_sub-block_and_block_for_both_variables
              add_sub-blocks_to_dependency_list
    endelseif
endelseif
```

Here each instruction or quad in a sub-block SB_i in basic block B_p is scanned and analyzed for its AFTER. If instruction has got Φ function in its RHS, then there would be AFTER dependencies coming from the predecessors of basic block B_p as per definition of phi function. While observing every predecessor B_q, dependency edges are created between the sub-blocks SB_iB_p and those sub-blocks SB_jB_q by adding SB_iB_p to the dependency list of SB_jB_q in dependency matrix.

A non Φ instruction can have one Use which is not defined in sub block SB_iB_p indicating that it is defined in one of its predecessor sub block SB_jB_q and hence the existence of dependency between SB_iB_p and SB_jB_q. The graph is traversed upwards covering all predecessor sub-blocks SB_jB_q for finding the AFTER dependency. For each dependency, one edge is added and the dependency matrix is updated. Similar action is taken for the case when RHS of quad contains two variables, both of which are defined in predecessors of the given basic block.

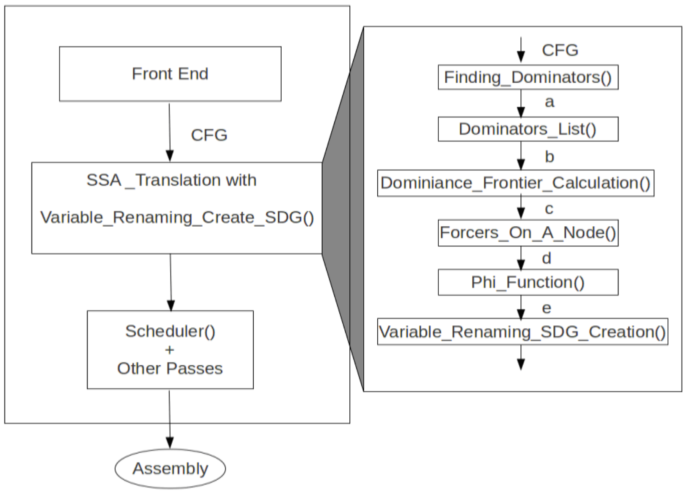


Figure 21. Modified Flow of Compiler to Create Disjoint Sub-blocks and SDG During SSA Translation

5.3 Compile Time Analysis of Approach 1 and Approach 2

This section analyses the compilation time requirement and algorithmic complexity of the proposed algorithms. Two different approaches to create disjoint sub-blocks and SDG is discussed.

A code has been developed to translate a non SSA form of the program to SSA form program. The code uses functions for finding dominators, dominance frontier, inserting Φ function and variable renaming.

The function to find dominators, dominance frontier and inserting Φ function are used in both the approach used for creating disjoint sub-blocks. Let the time taken to run these function be T1, T2 and T3 then T1+T2+T3 would be a common time factor in both approaches.

Let T4 be the time taken by variable_renaming( ) function. T4 is much greater in comparison to T1 and T2 as variable renaming is done for each variable belonging to every basic block, where as the operations, find_dominator and dominance frontier are done at basic block level. T4 is also larger than T3 even though the function to insert Φ is performed at instruction level as Φ function insertion is carried out in selected basic blocks only .

In the first approach to create sub-blocks, all basic blocks of CFG is traversed by making one extra pass [1]. In this pass, true dependency (RAW) between the instructions is computed inside each basic block and disjoint sub-blocks are created. Let the time taken for creating the disjoint sub-block be T5 and the time taken perform this task is almost same as time spent for variable renaming i.e., T5≥T4. The overall compilation is equal to summation of time T1, T2, T3, T4 and T5. Since T4 and T5 are much greater than T1, T2 and T3, the total compilation time can be considered to be proportional to summation of time T4 and T5.

In second approach for creating disjoint sub-blocks, the time taken for the variable renaming (T4) will remain same as renaming of variables are still performed by analyzing each instruction. The variable renaming and disjoint sub-blocks creation operations are done simultaneously by eliminating T5 factor. If T4' is the new time for performing variable renaming and disjoint sub-block creation, it is slightly more than T4. The overall compiler time dedicated for these operation is reduced to summation of time T1, T2, T3, and T4'. Since T4' is much greater than T1, T2 and T3, the total compilation time can be considered to be proportional to T4'.

The first approach to create SDG requires extra pass to check the AFTER dependency and insert edge between the two dependent sub-blocks. Let the time taken to perform this pass be T6. The second approach to create SDG does not require extra pass as operation to create the SDG is carried out at the time of variable renaming operation causing slight increase in variable renaming time from T4' to T4''. The overall compilation time to create SDG in second approach is summation of time T1, T2, T3, and T4''.

To discuss the complexity of the algorithms, consider the CFG with N basic blocks, E number of edges. Let T be the total number of ordinary assignment and total number of Φ functions, |DF| be total size of dominance frontier, and V be total number of variables. The complexity of SSA generation algorithm is linear O(E+T+|DF|), finding dominator has complexity O(E), finding DF's for CFG is $O(E+\sum_n |DF(n)|)$, Inserting Φ function is $O(\sum_n (T^*|DF(n)|)$, and variable renaming is O(V).

The algorithmic complexity of functions involved in disjoint sub-block creation such as makeSet, Union and Find set are as follows. The makeSet can be performed in constant time so it is O(1). The union operation is $O(N^2)$ or O(NlogN), depending on the size of the sub-block with other sub-block is getting merged. If larger sub-block is merged with smaller sub-block then it is O(NlogN) otherwise it is $O(N^2)$. The complexity of groupSet operation is O(N). Thus the overall complexity to create disjoint sub-blocks is $O(N^2)$.

To create SDG from the sub-blocks in N different basic blocks first AFTER dependency is computed and edge is inserted between the dependent sub-blocks. The algorithmic complexity of these two function is $O(N^4)$ in worst case when iterated on each basic block. By performing good ordering on sparse set of basic blocks these operations can be performed in $O(N^2)$. These two processes are common to both the approaches to create SDG described in Section 5.2.1 and Section 5.2.2 respectively.

5.4 Conclusion

The techniques to create the sub-blocks / SDG is discussed in this chapter. The schedule for these sub-blocks in the basic blocks and the SDG are created by local scheduler and global scheduler respectively. The implementation details of these schedulers are explained in Chapter 6 and Chapter 7.

CHAPTER 06 Intra Block Scheduling

The Intra block scheduling is an approach to schedule the sub-blocks belonging to a basic block. This chapter discusses the proposed intra block scheduling. The scheduler is termed as local scheduler or intra block scheduler. The intra block scheduler uses the disjoint sub-blocks created by fine grained thread extractor module to produce schedules for multiple cores of a multicore processor. The sub-blocks within the basic block are scheduled on different cores in a manner that balances the overall execution time and power consumption.

6.1 Introduction to Intra Block Scheduling

The problem of scheduling for parallel architecture by minimizing the overall execution time has been proven to be NP-Complete problem. A number of heuristics have been proposed with a view to find an optimal schedule that results in reduced execution time. The proposed intra block scheduler uses bin-packing approach to schedule the sub-blocks. The number of bins used are taken to be equal to the number of cores. The bin-packing problem is a NP-complete problem, an approximation algorithm with approximation factor of 2, with small running time O(n *log*n) is used. This algorithm runs in time O(n*log*n+n*c), where c is number of cores and n is number of sub-blocks.

Let $S_p = \{SB_1, SB_2, SB_3, \ldots SB_n\}$ be the list of sub-blocks belonging to basic block B_p. Each sub block can have varying number of SSA form instructions. For each sub-block SB_iB_p belonging to a basic block B_p, sub-block identifier and sub-block size information is maintained. Sub-block size is the number of instructions ***Ic*** in the sub-block SB_iB_p. The sub-blocks are sorted in non increasing order of their size so that the sub-block with higher size is given higher priority for scheduling.

Before the creation of schedule, at times sub-block merging is required. The two or more sub-blocks are merged to form a bigger sub-block which is named as hyper sub-block. The hyper sub-blocks are created to achieve below mentioned objectives.

i. While splitting basic blocks into sub-blocks, sometimes, we get number of sub-blocks with small number of instructions which use a small subset of available registers, leaving large pool of registers unutilized. The hyper sub-block creation results in efficient utilization of available register set. However, if the number of sub-blocks is less than number of cores, hyper sub blocks are not created.

ii. Let T_{i}, T_{i+1}, T_{i+2}, T_{i+3}..... T_{i+n} be the execution time of a set of sub-blocks S_{i}, S_{i+1}, S_{i+2}, S_{i+3}.........S_{i+n}. If $T_i \geq T_{i+1} + T_{i+2} + T_{i+3}$.....+ T_{i+n}, then scheduling the sub-blocks S_{i+1},S_{i+2},S_{i+3}.........S_{i+n} on multiple cores may not benefit in terms of speed-up as the total execution time will remain T_i. Substantial power is utilized by all these cores as all cores are active. If the sub-blocks S_{i+1}, S_{i+2}, S_{i+3}.........S_{i+n} are merged and is executed on any one of the cores by making all other core idle, power consumption can be reduced without compromising with the speedup.

iii. The sub-blocks could also be merged in a manner such that each merged sub-block has equal or nearly equal number of instructions. With this when the schedule is created, all the cores finish execution almost at same time.

6.1.1. Sub-block Merging

The merging algorithm to create hyper sub-block is given in algorithm 8. The algorithm takes a list S_p containing sub-blocks belonging to basic block B_P as input. The sub-blocks are arranged in descending order of their size i.e., number of instructions in each sub-blocks. The algorithm creates the hyper sub-blocks by merging the sub-blocks. Let T_{IC} be the total number of instructions in basic block B_P, S_{ICi} and R_{req_i} be the instruction count and register requirement of sub-block SB_iB_P.

The bins are used for creating the hyper sub-blocks. The hyper sub-block HSB_jB_P is created using j^th bin by merging one or more sub-blocks belonging to the basic block B_P. R_{avl_j} is total number of registers available for j^th bin which is initialized to number of registers in a core. R_{avl_j} is updated whenever a new sub-block is inserted into the bins or merged with the existing sub-blocks in the bin. The value of R_{avl_j} is modified by subtracting the register requirement of i^th sub-block (R_{req_i}). The proposed algorithm initially begins by creating two bins and populating it with first two sub-blocks from the list S_p. After this initialization, the algorithm picks up next sub-block from the list S_p and tries to merge it with sub-block present in any of the bin satisfying the first merge condition listed above. For all j bins, if R_{avl_j} is zero or less then the R_{req_i} of the sub-block then the sub-block cannot be merged with sub-blocks present in any of the bins then one more bin is created and sub-block is placed in it. The process is repeated for all the sub-block in the list. For ensuring the second and third condition of the algorithm, the bins are arranged in descending order of number of instructions in each bin in the beginning of every iteration. The basis for selecting two bins is the fact that multicore processor will have at least two cores. The first fit bin strategy is followed in the proposed algorithm, where a sub-block is merged with the first available bin which satisfies merge conditions. The sub-block list S_P is arranged in descending order to simplify the merge operation. The worst case algorithmic complexity of the merge algorithm with n sub-blocks in basic block B_P is O(n*log n) as smaller size sub-blocks are merged with bigger size sub-blocks in the bin (hyper sub-blocks). The hyper sub-blocks are sorted in descending order in the last step to reduce the search time of the scheduler while creating schedules.

Algorithm 8: Sub-block Merging

```
mergeSubblock($S_p$, no_of_sub-blocks, num_of_quads)
begin
    No_bins=2;
    bins[i] = 0 for all bins initially;
    binsize[i]= 0 for all bins initially;
    $R_{Avlj}$ = register in each core.

    for each sub-block $SB_iB_p$
    repeat
        Arrange bins in descending order of size

        for each bin j=1 to No_bins
        repeat
            if ($Ravl_j$ of $HSB_jB_P$ > $Rreq_i$ of $SB_iB_p$ )
            then
               break inner for loop
            end if
        end for
        if  j is same as No_bins
        then
            bins[j]= $SB_iB_p$;
            No_bins++;
        else
            Call **Union** (S,bins[j], $SB_iB_p$)
        $Ravl_j$ of $HSB_jB_P$ = $Ravl_j$ of $HSB_jB_P$ - $Rreq_i$ of $SB_iB_p$
    end for
End
```

An example depicting the merge operation is given Figure 22. The list S_P consists of ten disjoint sub-blocks with T_{Ic} = 57. The steps in creating hyper sub-blocks using proposed merging algorithm is shown in Figure 23 for a multicore processor with each core having four registers. Initially $Ravl_j$ value of the j^{th} bin to create the hyper sub-block HSB_jB_P is initialized to four.

List S_p	SB_8B_P $S_{IC_8}= 12$ $Rreq_8=4$	SB_2B_P $S_{IC_2}= 10$ $Rreq_2=5$	SB_9B_P $S_{IC_9}= 6$ $Rreq_9=2$	SB_4B_P $S_{IC_4}= 5$ $Rreq_4=2$	SB_1B_P $S_{IC_1}= 5$ $R_{req1}=2$	$SB_{10}B_P$ $S_{IC_{10}}= 4$ $Rreq_{10}=2$	SB_6B_P $S_{IC_6}= 4$ $Rreq_6=2$	SB_5B_P $S_{IC_5}= 4$ $Rreq_5=2$	SB_3B_P $S_{IC_3}= 4$ $Rreq_3=2$	SB_7B_P $S_{IC_7}= 3$ $Rreq_7=2$

Figure 22. Sub-blocks of Basic Block B_P

j Bins to create Hyper Sub-blocks HSB_jB_P	HSB_1B_P	HSB_2B_P	HSB_3B_P	HSB_4B_P	HSB_5B_P	HSB_6B_P
Iteration 1	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$				
Iteration 3	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P $H_{IC_3}= 6$ $Ravl_3=2$			
Iteration 4	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$			
Iteration 5	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $H_{IC_4}= 5$ $Ravl_4==2$		
Iteration 6	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $SB_{10}B_P$ $H_{IC_4}= 9$ $Ravl_4=0$		
Iteration 7	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $SB_{10}B_P$ $H_{IC_4}= 9$ $Ravl_4=0$	SB_6B_P $H_{IC_5}= 4$ $Ravl_5=2$	
Iteration 8	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $SB_{10}B_P$ $H_{IC_4}= 9$ $Ravl_4=0$	SB_6B_P SB_5B_P $H_{IC_5}= 8$ $Ravl_5=0$	
Iteration 9	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $SB_{10}B_P$ $H_{IC_4}= 9$ $Ravl_4=0$	SB_6B_P SB_5B_P $H_{IC\ 5}= 8$ $Ravl_5=0$	SB_3B_P $H_{IC_6}= 4$ $Ravl_6=2$
Iteration 10 (Final Hyper sub-blocks)	SB_8B_P $H_{IC_1}= 12$ $Ravl_1=0$	SB_2B_P $H_{IC_2}= 10$ $Ravl_2=0$	SB_9B_P SB_4B_P $H_{IC_3}= 11$ $Ravl_3=0$	SB_1B_P $SB_{10}B_P$ $H_{IC_4}= 9$ $Ravl_4=0$	SB_6B_P SB_5B_P $H_{IC_5}= 8$ $Ravl_5=0$	SB_3B_P SB_7B_P $I_{C6}= 7$ $Ravl_6=0$

Figure 23. Steps in Merging Sub-blocks of Basic Block B_P

The algorithm begins with creating two bins to create hyper sub-blocks HSB_1B_P and HSB_2B_P using the sub-block SB_8B_P and SB_2B_P respectively. In the second iteration a new bin to create hyper sub-block HSB_3B_P is created for the sub-block SB_9B_P as the register availability of the bins having hyper sub-block HSB_1B_P and HSB_2B_P is zero, i.e,. $Ravl_1=0$ and $Ravl_2=0$. The sub-block SB_4B_P is merged with the sub-block SB_9B_P in the bin which is creating hyper sub-block HSB_3B_P as its merge condition is satisfied. We can observe that the size of all three bins in fourth iteration are almost equal. This process is repeated for the other sub-blocks. At the end of 9th iteration six hyper sub-blocks are created by the merge algorithm. These six hyper sub-blocks are sorted in descending order of their size and are used by scheduler to create schedules for multicore processor.

6.1.2. Intra Block Scheduler

The hyper sub-blocks created by the sub-block merge module is taken as input to create schedule for the multicore processor. The execution time TStc and available time TAvl of each is core is maintained to create schedule. Each hyper sub-blocks in the ready queue is scheduled on to the core which has minimum TStc at that instance. After scheduling the hyper sub-block HSB_iB_p on to the core its TStc is updated.

Algorithm 9: Intra Block Scheduler

```
scheduleBlock(Basic Block $B_p$, Hyper sub-block List HSList)
begin
  for
  for each core i
  repeat
    TStc[i]=0; TAvl[i]=0;
  end for
  for each hyper sub-block HSB in HSList in sorted order
  repeat
    find min_core such that TStc[min_core] is minimum among all cores
    Schedule the Hsub-block HSB on to min_core
    TStc[min_core]= TStc[min_core] + HSubBlockSize[HSB]
  end for
end
```

Let HSList be the list of hyper sub-blocks created by merging the sub-blocks in the basic block B_p. The hyper sub-blocks are sorted in non-increasing order of their size. If L is the number of hyper sub-blocks in the HSList and N is the number of cores in the processor. The scheduler first check if the $L \leq N$. The scheduler creates L schedules if the condition is true. Otherwise the scheduler creates M schedules for N cores where is $M \leq N$. In the first iteration the scheduler creates N schedules for N cores. In the next iteration it checks if it can create schedules for N-1 cores without compromising with the speed-up. If it is possible it will create schedules for N-1 cores. Similarly it will keep reducing the number of cores till speed-up of two consecutive iteration are same and choose the value of iteration before failure as M and creates M schedules. This exercise is done to utilize only required number of cores in order to reduce power consumption and to increase performance per power.

The Schedules created for Dual core, Quad core and three active core processors by Intra block scheduler is shown in Figure 24. When the hyper sub-blocks are scheduled, it takes 29, 19 and 19 clock cycles against 32, 22, and 22 clock cycles on Dual core, Quad core and Three active core respectively when sub-blocks are scheduled without merging. The difference between three active core and quad core processor is that that performance per power of three active core will be more than quad core processor.

Dual Core		Quad Core				Three Active Core		
HSB_1B_P HSB_4B_P HSB_6B_P TSct=28	HSB_3B_P HSB_2B_P HSB_5B_P TSct = 29	HSB_1B_P TSct =12	HSB_2B_P HSB_4B_P TSct = 19	HSB_3B_P HSB_5B_P TSct = 19	HSB_6B_P TSct = 7	HSB_1B_P HSB_6B_P TSct =19	HSB_2B_P HSB_4B_P TSct = 19	HSB_3B_P HSB_5B_P TSct = 19

Figure 24. Schedules Created by Intra Block Scheduler

6.2 Results

The result and observations on execution of benchmark programs when scheduled using proposed Intra block scheduler is discussed in this section. The equations used for computing the results are explained in Section 3.4. The relative results in terms of speed-up, power consumption and performance per power achieved by executing benchmark programs on dual core and quad core processor is shown.

The result in Figure 25 shows that the speed-up increases as number of cores increase, which makes it evident that all the cores are utilized towards achieving maximum gain. The gain in speed-up was the expected output from multicore processor but was not obvious for the non multithreaded applications.

The speed-up decreases when the same code is run on 3 active cores, however the performance per power improves as is shown in Figure 27.

The power consumed to execute the test cases is captured and performance per power of each test case is calculated as shown in Figure 26 and Figure 27 respectively. It is observed that power increases as the number of cores increase. The power consumption is lower when 3 cores are used instead of 4 cores. Thus the performance per power of quad core machine is higher when 3 cores are used instead of 4 cores. The effect of using 3 cores by slightly compromising with speed-up is shown in Figure 25. The communication costs of the proposed algorithms are shown in Figure 28.

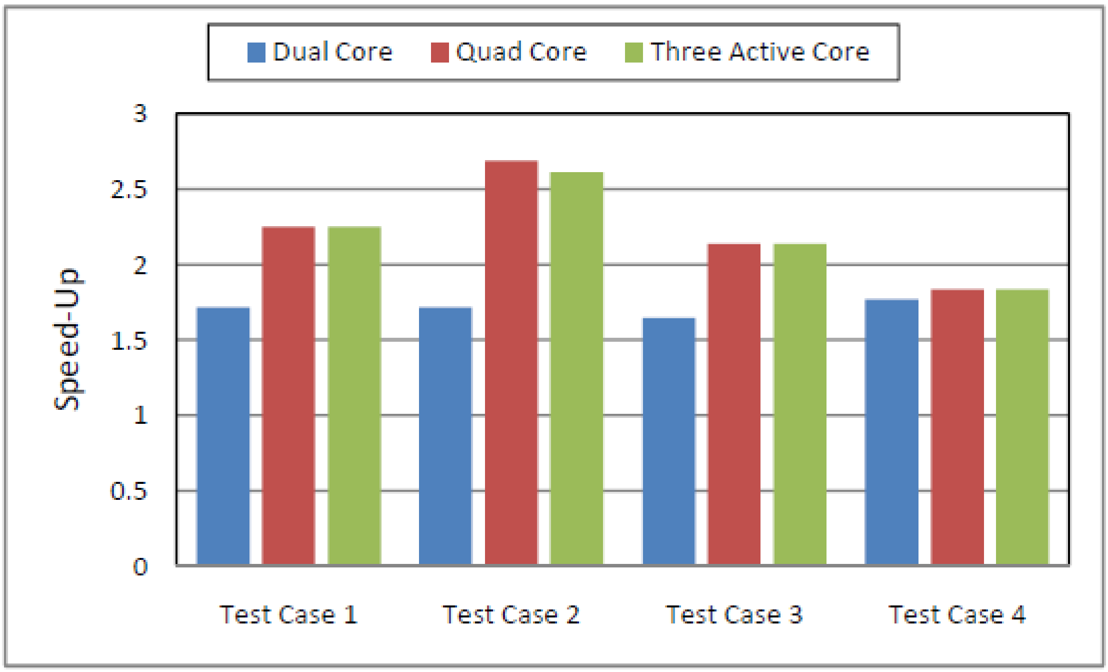


Figure 25. Speed-up analysis For Intra Block Scheduling

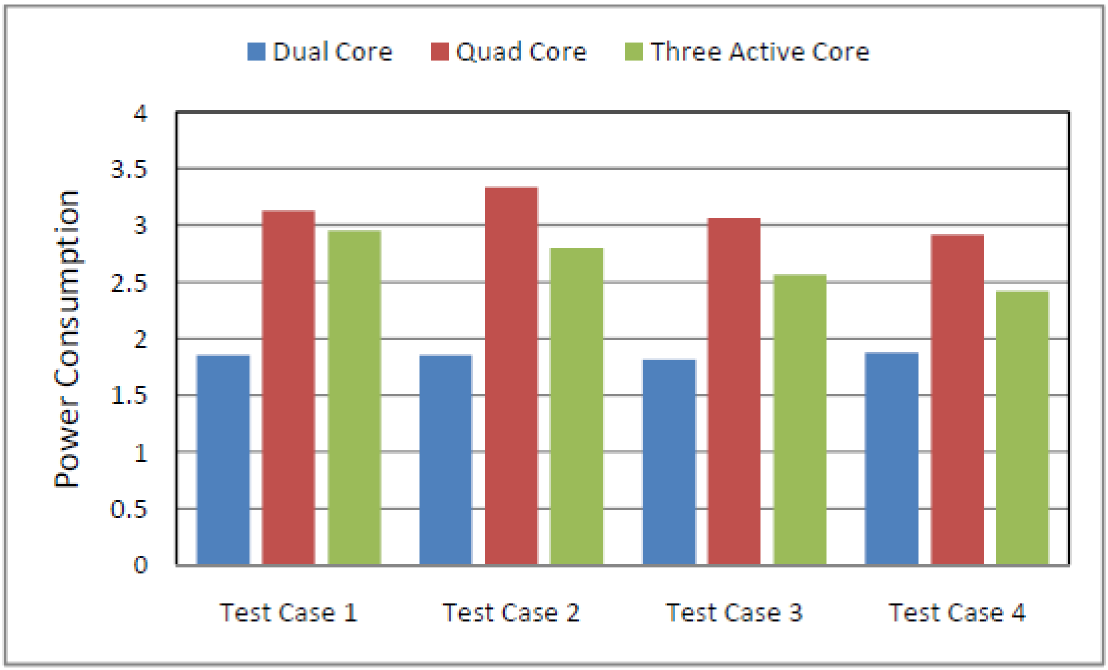


Figure 26. Power Consumption For Intra Block Scheduling

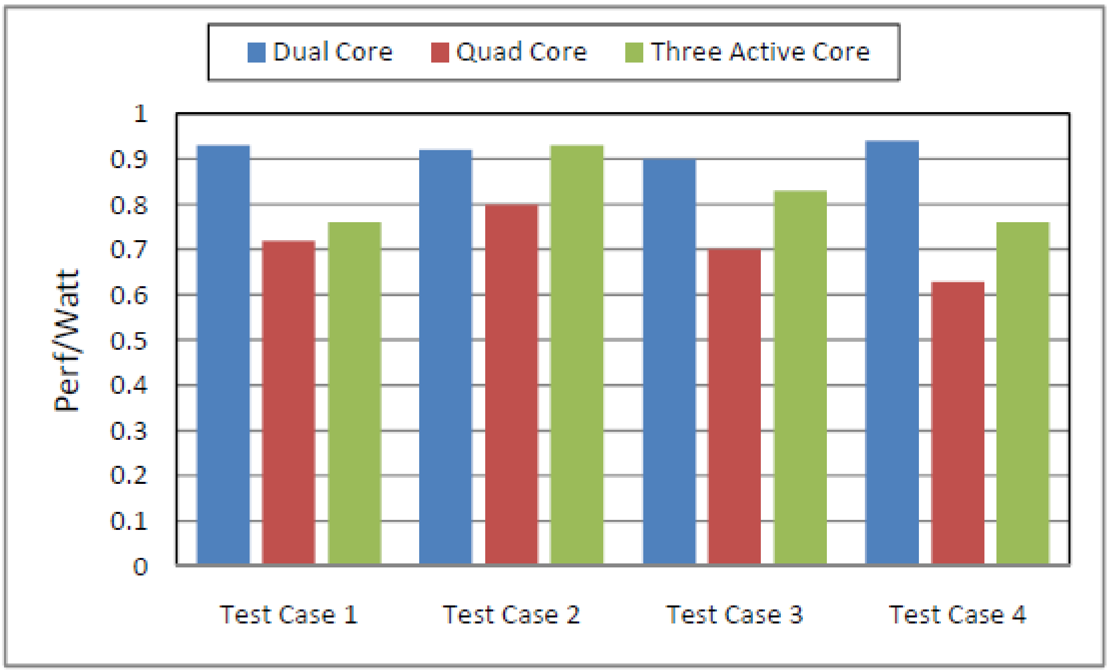


Figure 27. Performance / Power For Intra Block Scheduling

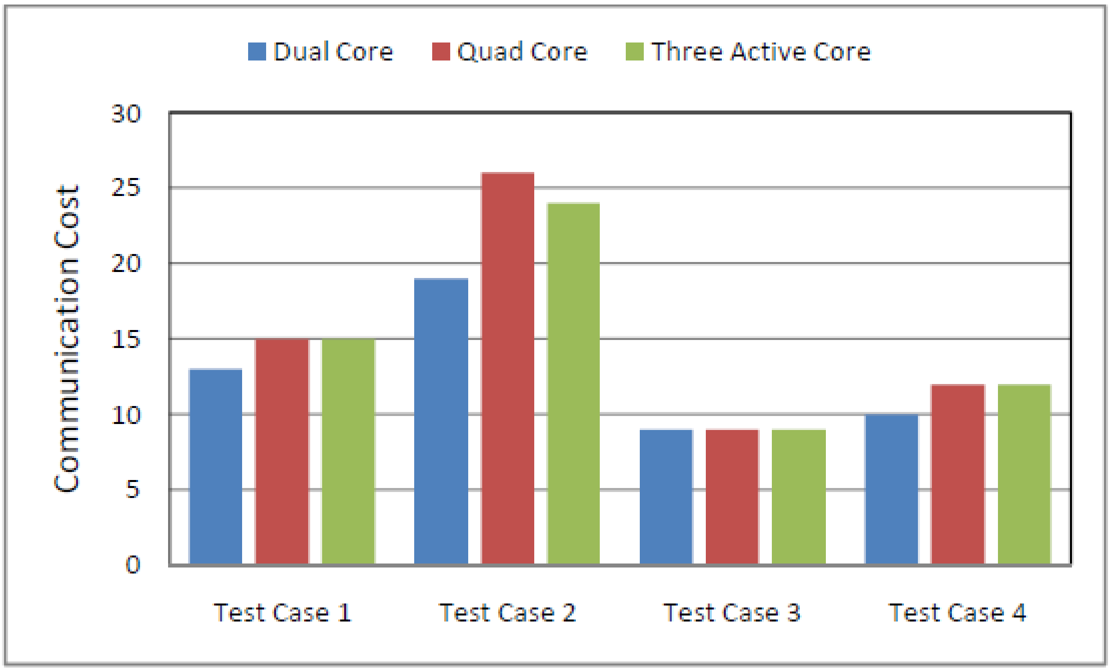


Figure 28. Communication Cost For Intra Block Scheduling

The Intra block scheduler (IBS) was designed for locally scheduling the disjoint sub-blocks. The IBS will not ensure that the dependent sub-blocks in rest of the basic block of CFG are scheduled on same core. This may lead to high communication and data movement between sub-blocks scheduled on different cores, which intern can have serious impact on speed-up and power consumption. To overcome these limitations Inter block scheduler is proposed in Chapter 07. The results of Intra block scheduler is compared with Inter block scheduler in Chapter 07.

CHAPTER 07 *Inter Block Scheduling*

The implementation details of global scheduler is given in this chapter. The global scheduler is termed as Inter Block scheduler. Similar to local scheduler (Intra Block Scheduler), the schedule generated by global scheduler for each core consists of list of sub-blocks. In contrast to local scheduler, the global scheduler identifies all the independent sub-blocks across the basic blocks in a CFG to formulate the schedule

In this chapter, four novel global scheduling heuristics are proposed. These heuristics are designed to obtain high performance, low communication cost, and high performance per power. The scalability issues with increasing number of cores are also been explored. The novelty of proposed algorithms is in its efficient scheduling strategies, which translates into improved performance without increasing the algorithmic complexity.

The proposed global scheduling algorithms uses the sub-block dependency graph (SDG) to schedule sub-block on to multiple cores. The first algorithm, called the Height Instruction Count Based (HIB) algorithm, is based on priority calculated using the height and instruction count of the sub-block in the SDG. It is a linear-time algorithm which uses an effective search strategy to schedule the sub-block on to the core with minimum schedule time. The second scheduling algorithm is based on dependency between sub-blocks and has been named as Dependent Sub-Block scheduler (DSB). All the paths from a given block to leaf node of SDG is identified and schedule latency for each path is computed. The sub-blocks in the path with highest schedule latency are chosen for scheduling on different cores. The third algorithm is Maximum Dependency Sub-block First (MDSF). It calculates the priority of the sub-block based on maximum dependencies and minimum execution time. The fourth algorithm is Longest Latency Sub-block First (LLSF) which schedules considering only latency of the sub-block. The proposed algorithms have been evaluated through extensive experimentations and results have been compared with existing algorithms.

In General, the global scheduler selects the sub-block SB_iB_p from the sub-block dependency matrix if dependency list of SB_iB_p is empty. Once SB_iB_p is scheduled and completes its execution, it is removed from all dependency lists. The scheduler uses the values of *height* ($Height_i$) of sub block in SDG, predicted *finish time* (TFns), *ready time* (TRdy), schedule latency (L_i) of a sub-block, and *total schedule time of core* (Tsct) and are computed using equations 6, 7, 8, 9, and 10 respectively.

7.1 Height-Instruction Count Based (HIB)

The HIB scheduler uses the sub-block dependency graph represented in the form of matrix to take scheduling decision. The scheduler creates a priority queue using SDG, and schedules the sub-blocks on to multiple cores. Scheduler will schedule the sub-block with highest priority in the priority queue on the core with minimum Tsct. The scheduler updates the priority queue with new sub-blocks and remove their entry from dependency matrix. A sub-blocks can be added in priority queue if dependency list of that sub-block is NULL. Priority of the node is computed based on height ($Height_i$) and instruction count. The node at highest level and more instruction count is given highest priority.

Algorithm 10 : Height Instruction Count Based Scheduler

```
Calculate height of each sub-block
    Height of sub-block i= max( height of all immediate successors) + 1)
Initialize a Priority Queue
   Q={All head node i.e nodes having only out going edges}
Schedule:
 If single core
i. Remove highest priority node from queue.
ii. Insert those nodes which are ready to schedule after scheduling this node.
iii. Schedule the node on core.
iv. Repeat same Process until queue gets empty.
 If multiple cores
i. Repeat steps ii) to v) until queue gets empty.
ii. Select a core with minimum schedule time.
iii. Select a node with highest priority (see above).
If ready-time of all nodes present in queue is greater than current core schedule
time then insert 1 free cycle.
Goto step ii)
iv. Schedule node on core and increment current core time.
v. Update finish time of this node and reay-time of all its immediate successor.
vi. Place its immediate successors in queue if they are ready to schedule (see above)
and revise the priorities of old nodes according to the priorities for new nodes.
Goto step i)
vii. END.
```

7.2 Dependent Sub-block Based (DSB)

The DSB scheduler collects all the sub-blocks and stores them in non-increasing order of their schedule latency. Initially the scheduler picks the sub-block with the highest schedule latency and schedules it on to any one of the cores. Later scheduler picks the immediate ready successor of the previously scheduled sub-block in the SDG. The successor sub-block is scheduled on to the same core if the TSct of core is less than other core, otherwise it will switch to core with lowest TSct. After scheduling each sub-block, the

TSct of the core is updated. The advantage of scheduling dependent sub-blocks on to same core is that it results into reduced communication between the cores.

Algorithm 11: Dependent Sub-block Based Scheduler

```
 Find Latency
 Sort sub-blocks by descending latency.
Schedule :-
a. Single Core – in order of sorted list.
b. Multi Core :
i. Repeat steps (ii) to (viii) until list gets empty
ii. temp ← top (list) (ready sub-block)
iii. Schedule temp & increment schedule time of this core.
iv. Update finish-time of temp and ready-time for all immediate successors.
v. If any immediate successor of temp is ready (check in order of list) & list is non-empty
    temp ← immediate successor
    goto step (iii)
vi. If schedule time of current core is less than max schedule time & list is non-empty
   goto step (ii)
vii. Max schedule time ← schedule time of current core
viii. If list is non-empty    switch core
   goto step (ii)
ix. END
```

7.3 Maximum Dependent Sub-block First (MDS)

The scheduling decision of MDS algorithm is purely based on the structure of the SDG. The sub-block having maximum successors is given higher priority and is picked by the scheduler for scheduling it on to the core with least TSct. The MDS scheduler maintains the ready list. Priority of sub-block SB_iB_p in ready list is computed based on TRdy, TFns and its dependencies.

A sub-block SB_iB_p can be inserted into ready list if its dependency list is empty, i.e. all the sub-block on which SB_iB_p was depending have finished there execution.

Algorithm 12: Maximum Dependent Sub-block First Scheduler

```
1. Collect all the sub-blocks which are ready for execution.
2. Find out the priorities for all the sub-blocks in the ready list
3. Schedule :
Single Core:
i. Schedule the sub-block with highest priority on to the core.
ii. Update the adjacency matrix.
 Multi Core:
i. Find out the core which is free.
ii. Schedule the sub-block with highest priority to the core which is selected.
iii. Update the adjacency matrix.
4. Goto step 1
5. END
```

7.4 Longest Latency Sub-block First (LLSF)

LLSF scheduler is similar to the DSB scheduler except the choice made to schedule the successor sub-block on to the same core. The only choice of selecting sub-block is its schedule latency, The scheduler picks the sub-block with the highest schedule latency (L_i) and schedules it on the core with lowest TSct.

Algorithm 13: Longest Latency Sub-block First Scheduler

```
Find Latency
Sort sub-blocks by descending latency.
Schedule :-
Single Core – in order of sorted list.
Multi Core :
i. Repeat steps (ii) to (vii) until list gets empty
ii. temp ← top (list) (ready sub-block)
iii. Schedule temp & increment schedule time of this core.
iv. Update finish-time of temp and ready-time for all immediate successors.
v. If schedule time of current core is less than max schedule time & list is non-
empty
   Goto step (ii)
vi. Max schedule time ← schedule time of current core
vii. If list is non-empty switch core
   goto step (ii)
viii. END
```

This scheduler has advantage over other proposed global schedulers in terms of speed-up, but has penalty of communication cost and power. This scheduler can be used in environment where performance is crucial and no other optimization is required. Scheduler uses sorted list of sub-blocks. Sorting is based on descending order of scheduling latency. Each node in list contains sub-block and its respective latency.

7.5 Results

The main aim of the proposed work was to equally utilize all the available cores. The relative results in terms of speed-up, power consumption and performance per power achieved by executing benchmark programs on dual core and quad core processor is shown. The equations used for computing the results are explained in Section 3.4. The result in Figure 29 shows that the speed-up increases as number of cores increase, which makes it evident that all the cores are utilized towards achieving maximum gain. The speed-up decreases when the same code is run on 3 active cores, and at the same time per power performance improves as is shown in Figure 31. The power consumed to execute the test cases is captured and performance per power of each test case is calculated as shown in Figure 30 and Figure 31 respectively. It is observed that power increases as the number of cores increase. The power consumption is lower when 3 cores are used instead of 4 cores. Thus the performance per power of quad core machine is higher when 3 cores are used instead of 4 cores. The effect of using 3 cores by slightly compromising with speed-up is shown in Figure 29. This power optimization can be used in an environment where power is critical. The communication costs of the proposed algorithms are shown in Figure 32, and are compared with the communication cost of intra block scheduler.

The general observations on execution of benchmark programs when scheduled using proposed schedulers are as follows. These observation were the expected output from multicore processor but was not obvious for the non-multithreaded applications.

- Speed up increases as number of cores increase.
- Power consumption increases with increased utilization of cores..
- Performance per power decreases when more cores are used.
- Performance per power increases with 3 active cores compared to 4 active cores in a quad core machine with slight compromise on speed-up.
- Power increases and speed-up decreases when communication between cores increases.
- Performance per power increase when communication decreases.

Intra block scheduler (IBS) was designed for locally scheduling the disjoint sub-blocks within the basic block. But when whole program (CFG) is to be scheduled, sub-blocks in other basic blocks may communicate with sub-blocks scheduled on a different core. Thus communication cost is higher when intra block scheduler is used.

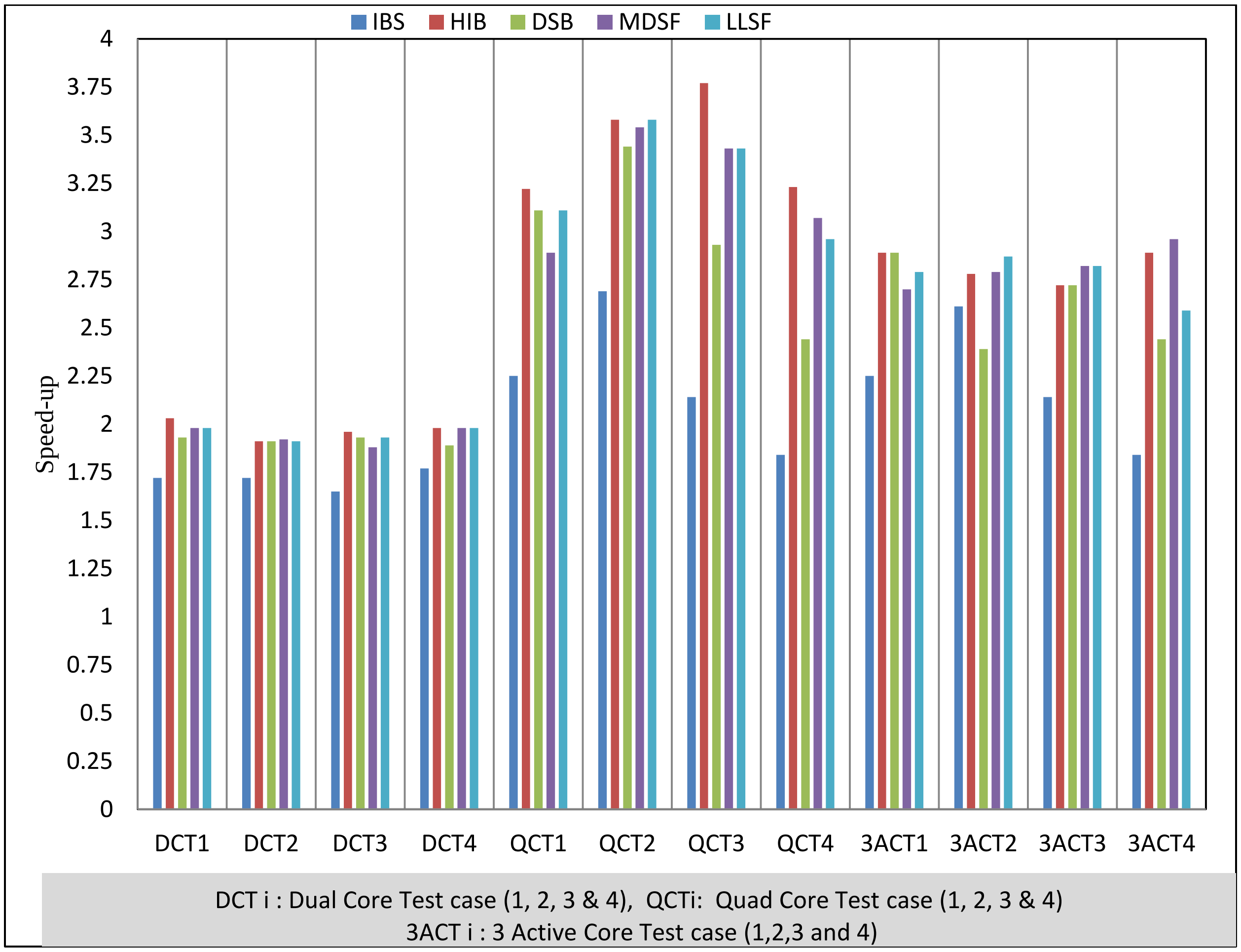


Figure 29. Speed-up Analysis For Inter Block Schedulers

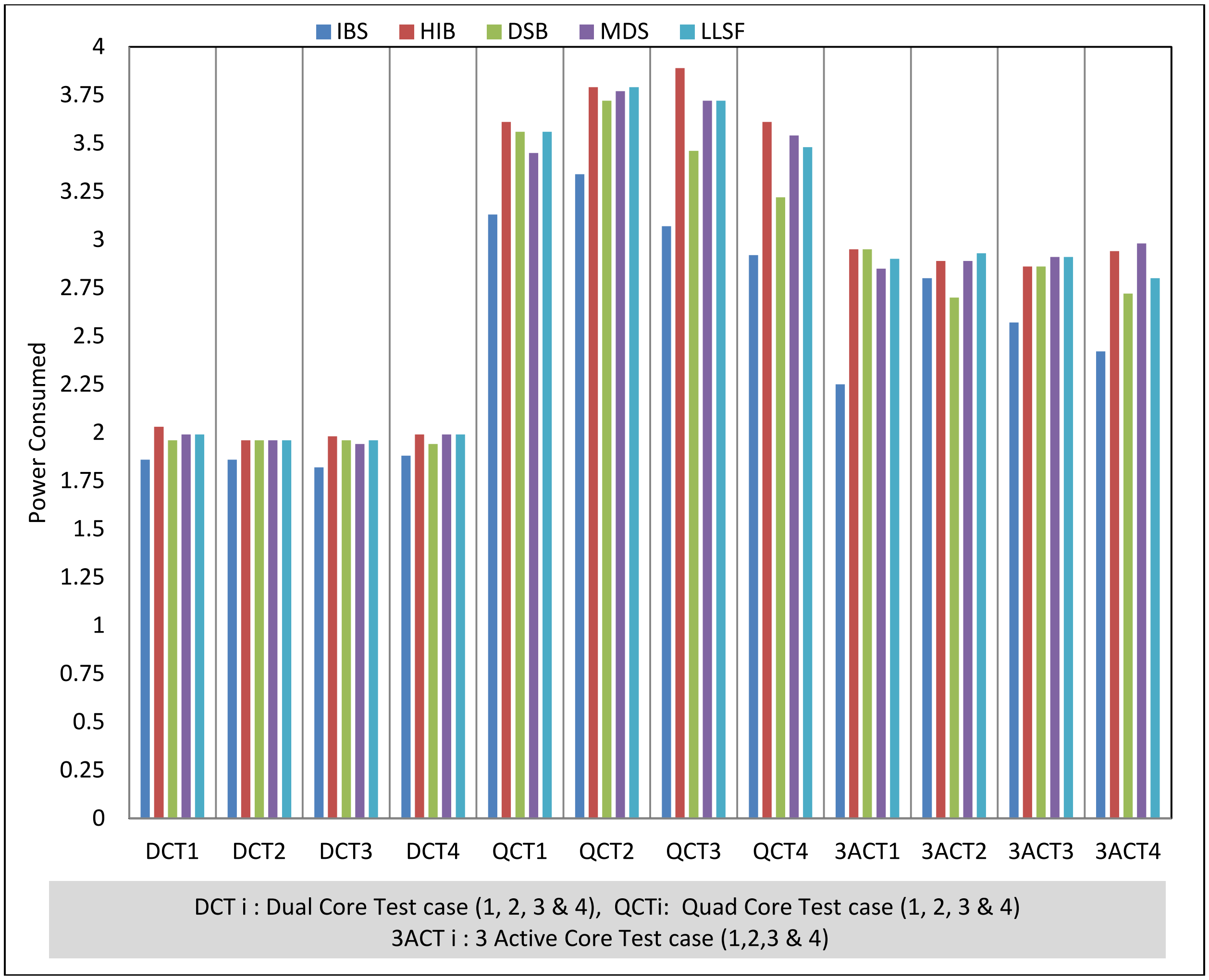


Figure 30. Power Analysis For Inter Block Schedulers

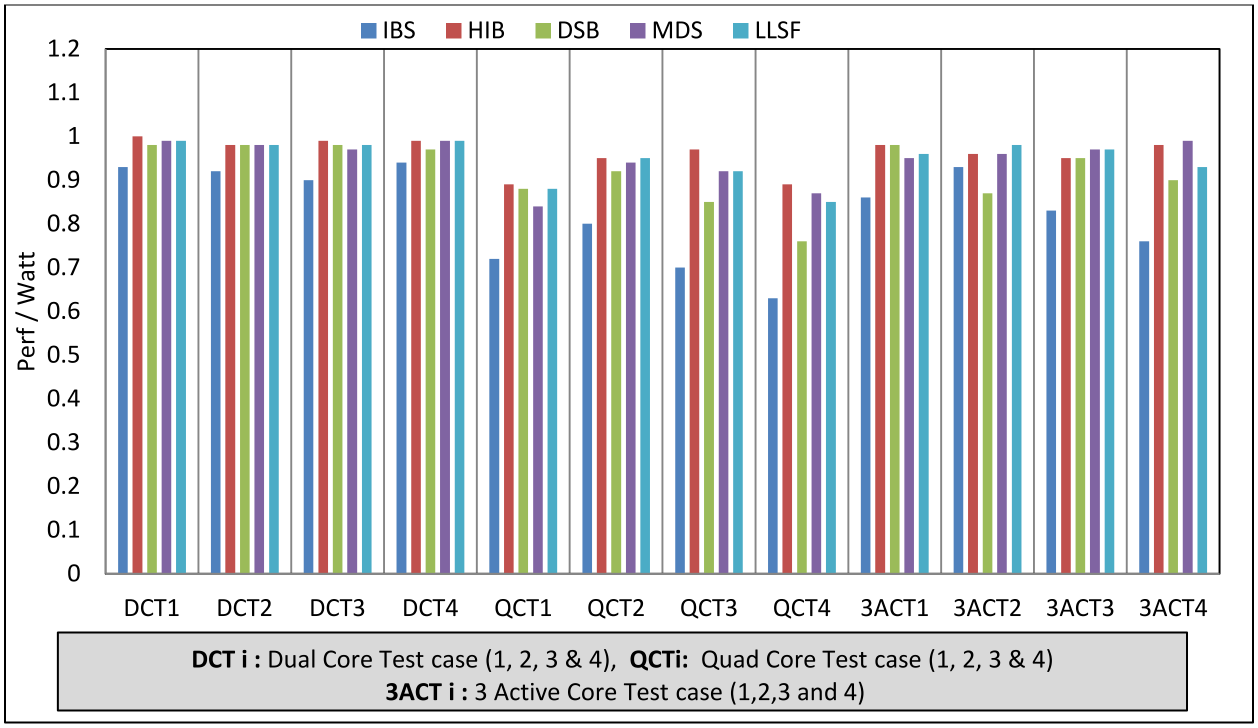


Figure 31. Performance Per Power Analysis for Inter Block Schedulers

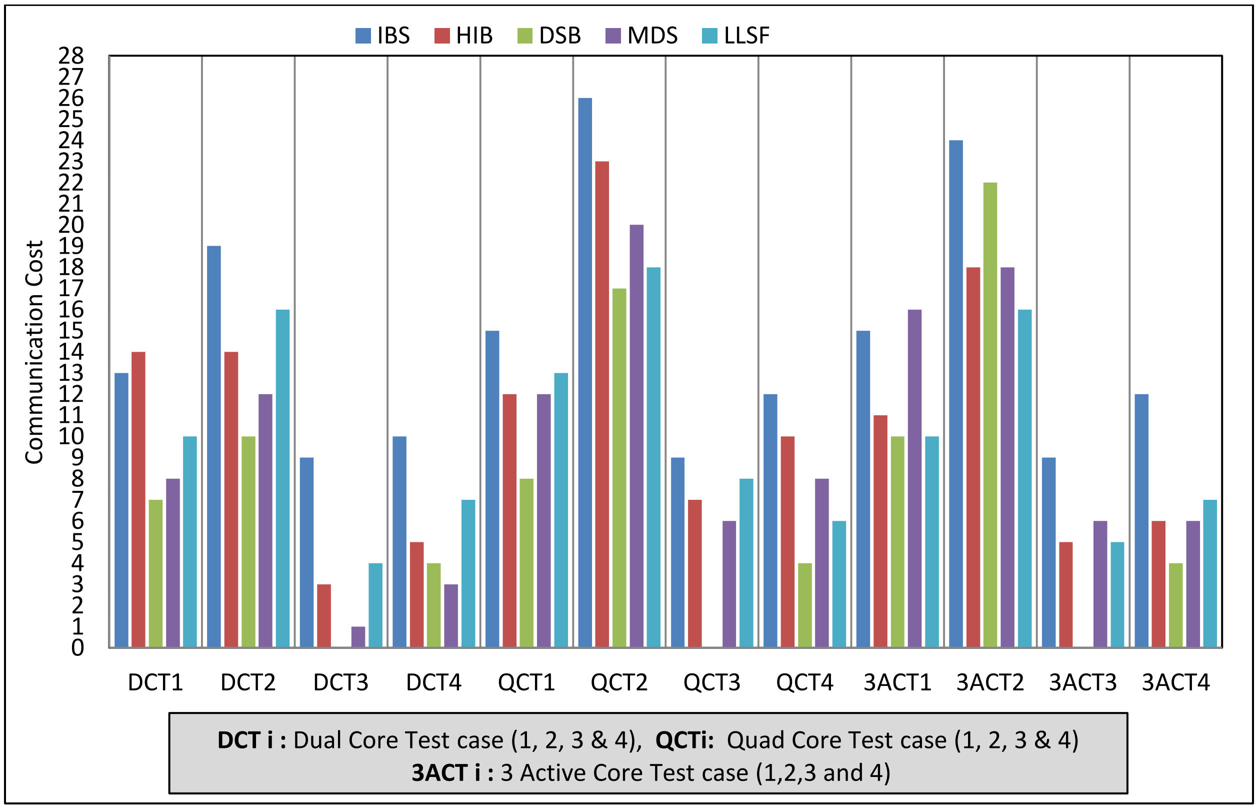


Figure 32. Communication Cost For Inter Block Schedulers

7.5.1 Discussion

This section discusses various results obtained using test case 1. In Figure 33, the speed-up and power consumed is compared. Power and performance per power are compared in Figure 34, and the speed-up and communication cost are compared in Figure 35. The same is applicable to other 3 test cases as is shown in Figure 36, Figure 37 & Figure 38 and other benchmark programs that are used to evaluate the proposed work.

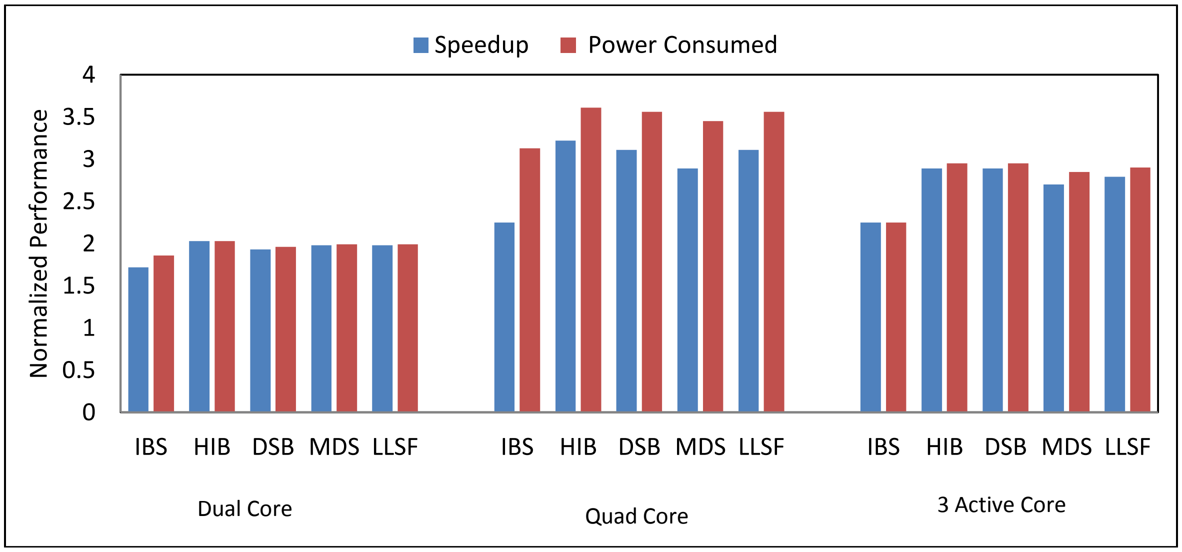


Figure 33. Speed-up vs Power Consumed

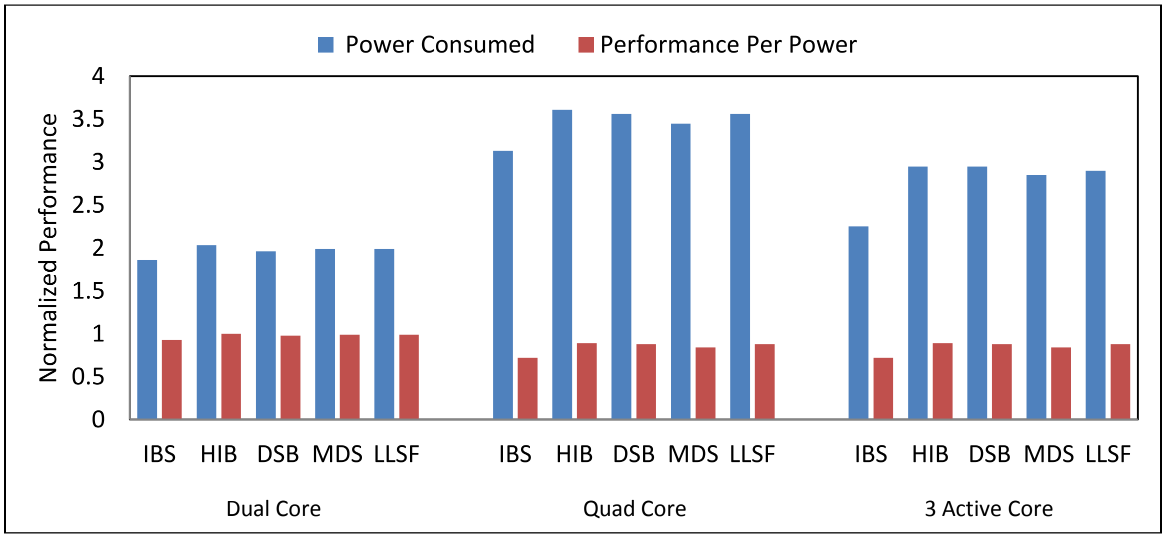


Figure 34. Power vs Performance Per Power

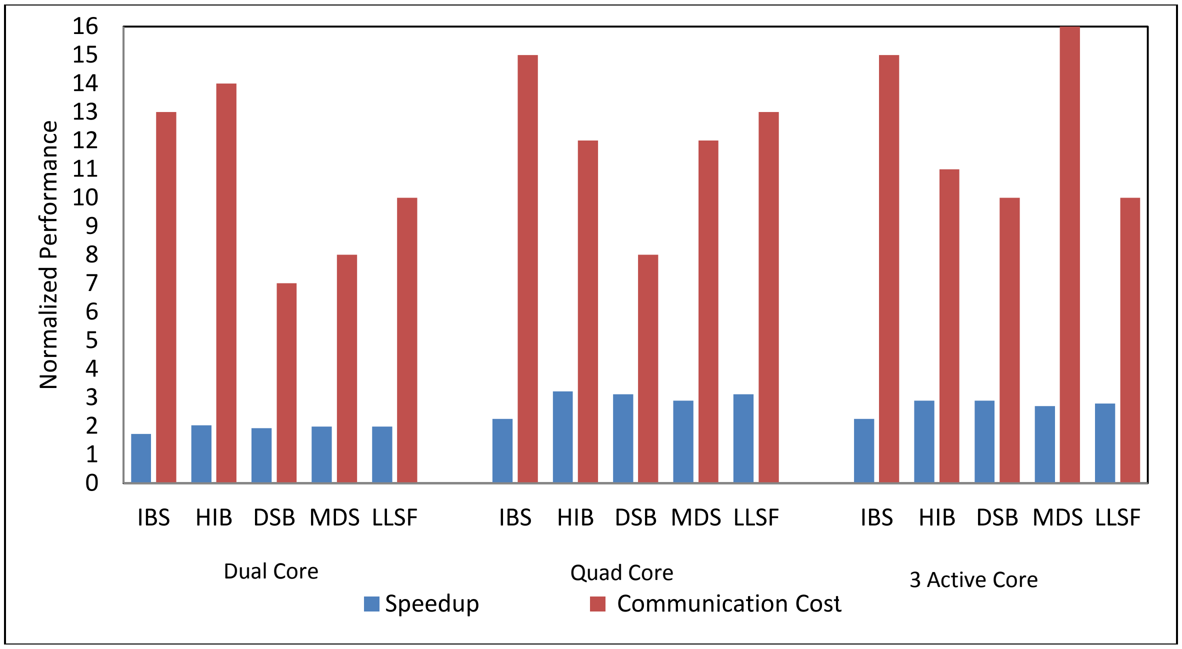


Figure 35. Speed-up vs Communication Cost

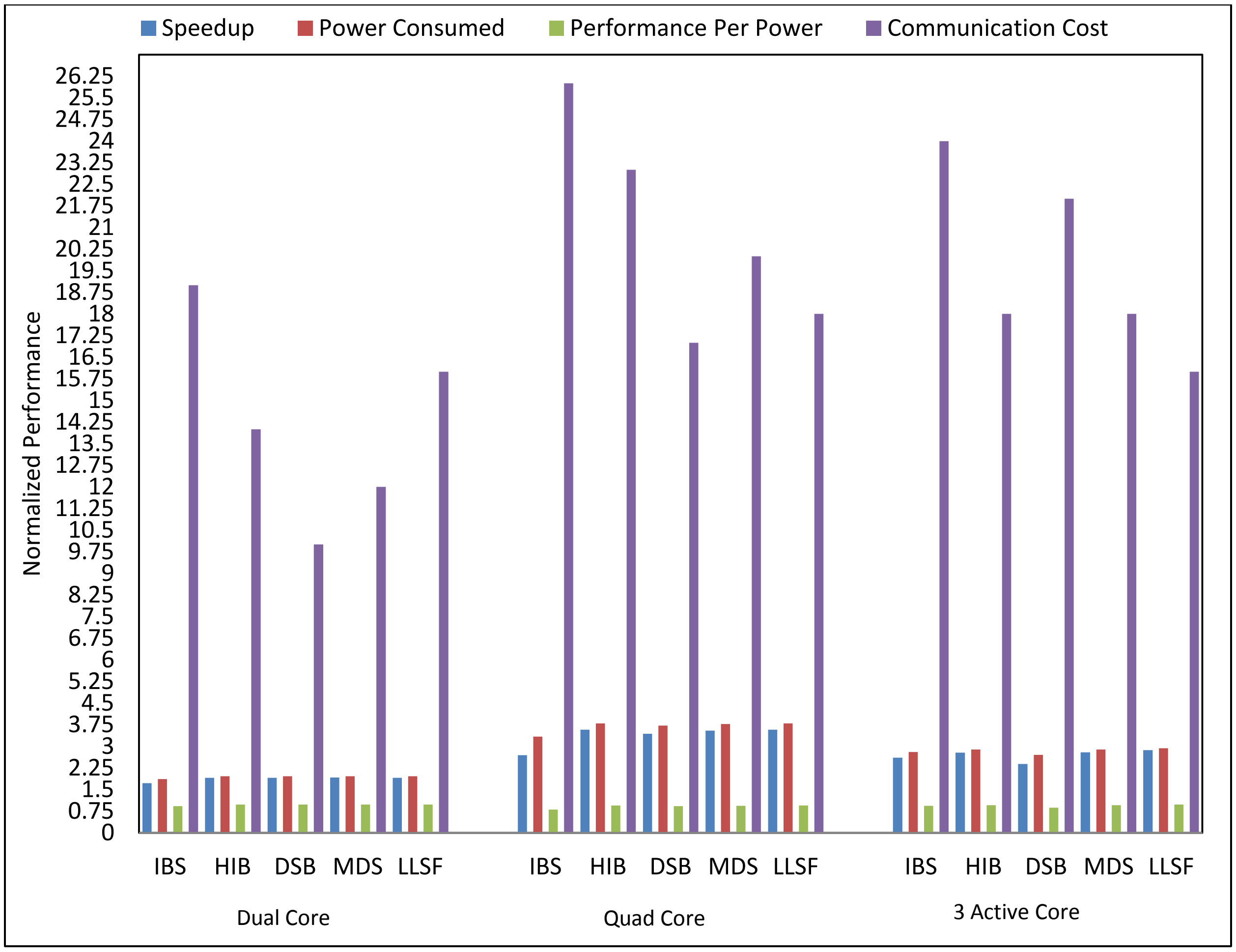


Figure 36. Analysis of Test Case 2

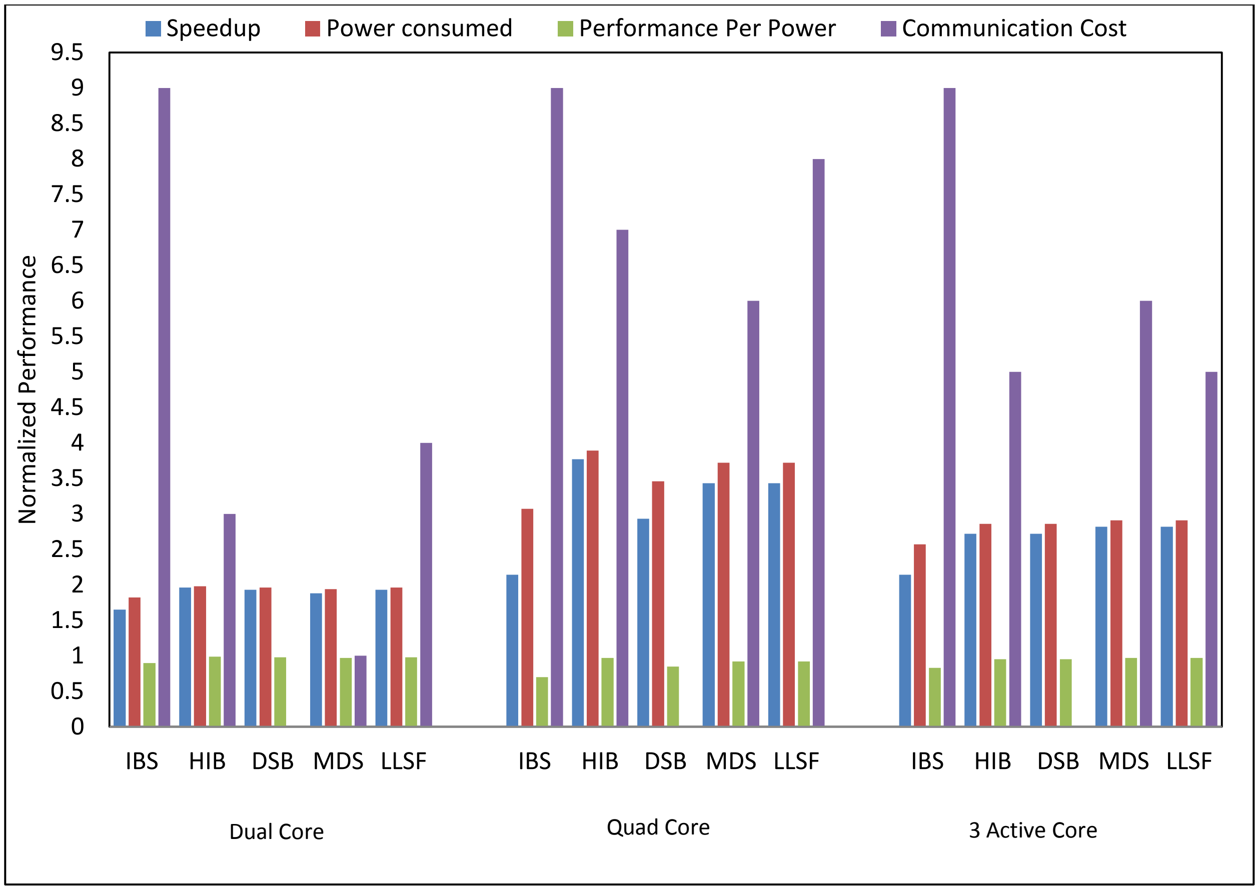


Figure 37. Analysis of Test Case 3

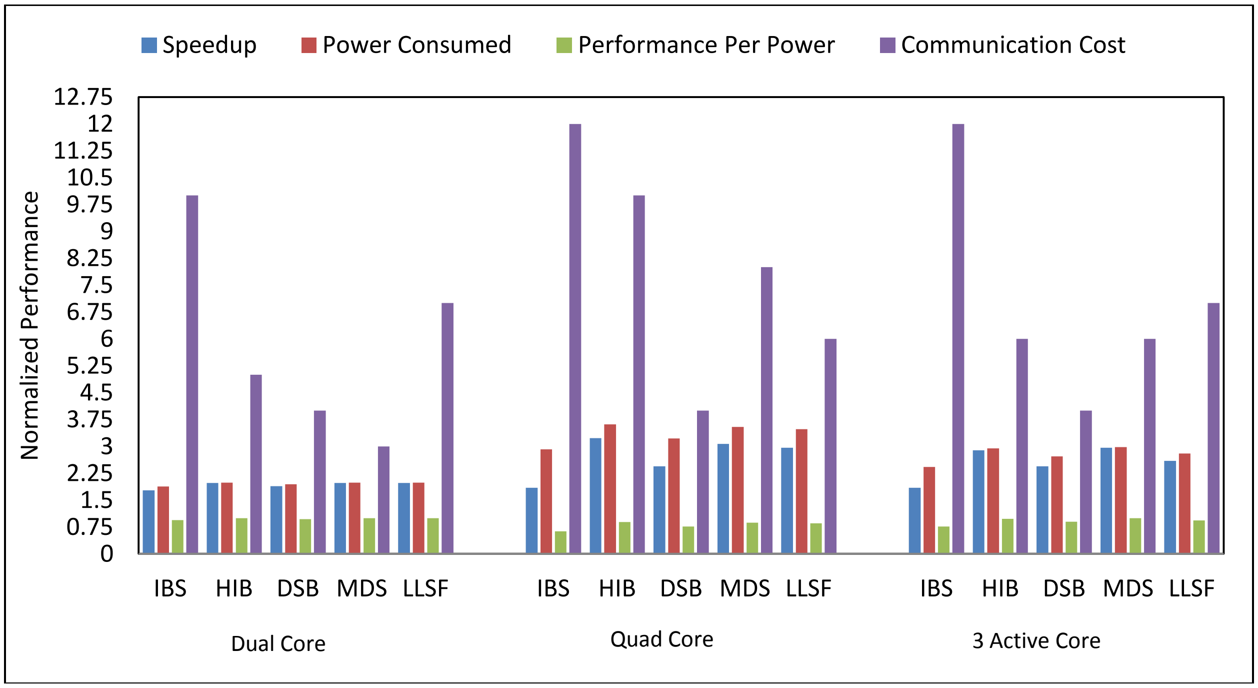


Figure 38. Analysis of Test Case 4

The speed-up gain of HIB scheduler is slightly high compared to other schedulers on all the machines. The communication cost of HIB scheduler is high which influence increase in power consumption and decreases the performance per power.

The dependent sub-block based (DSB) scheduler is made memory efficient by scheduling connected sub graphs in SDG, i.e. dependent sub-blocks on to same core. Thus the DSB scheduler reduces the communication between the cores. But, as number of core increases this heuristic suffers. This scheduler will not schedule the ready sub-block on the other idle cores, but would wait to schedule the ready sub-block on to the core on which its ancestor sub-block in SDG executed. This may lead to unbalanced scheduling. The DSB scheduler is suitable when SDG is dense and benefits only when less cores are used by applying power optimization technique.

Maximum dependent sub-block first (MDS) scheduler tries to balance the communication cost, power consumption and speed up. The values of speed up, power consumption and the communication cost lie between the values of those metrics achieved using HIB and DSB schedulers. Similar to DSB scheduler, MDS will suffer when SDG is dense and the scheduler doesn't apply power optimization.

Longest latency sub-block first (LLSF) will overcome the limitations of DSB and MDS schedulers. LLSF scheduler picks the sub-block with longest latency and schedules it on to the core with least execution time. The communication cost is less as compared to HIB scheduler but slightly higher than DBS and MDS schedulers. Thus the values for LLSF scheduler in terms of speed-up and performance per power gain lie between the HIB and DSB/MDS. This scheduler is scalable in terms of number of cores. LLSF when compared with HIB may not be compiler efficient as it perform linear search to find the sub-block with the longest latency.

7.6 Conclusion

The work in the chapter proposes various compiler level global scheduling techniques for multicore processors. The goal underlying these techniques is to promote extraction of ILP without explicitly specifying parallelizable fraction of the program by the programmer. To achieve this, the basic blocks of the control flow graph of a program are subdivide into the multiple sub-blocks and there by a sub-block dependency graph is constructed. The proposed schedulers, depending on sub-block dependency and their order of execution, allocate the sub-blocks in the dependency graph to multiple cores selectively. These schedulers also carry out locality optimization to minimize communication latency among the cores and to minimize the overhead of hardware based instruction reordering. A comparative analysis of performance and the inter-core communication latency has been presented. The results obtained thereof also indicate how these schedulers perform in terms of power consumption and the speed up achieved when the number of active cores varies. From the results it can be observed that a better and balanced speedup per watt consumption can be obtained. Though the results are shown for dual core, quad core and active 3 core processors, the proposed scheduler theoretically can scale to handle larger number of cores as the sub-block formation technique is independent of number of cores and memory access. Memory contention can have impact on scalability which would need to be further investigated.

CHAPTER 08 Register Allocation For Multicore Processor

Register allocation phase of compiler maps the unbounded number of program variables to a fixed number of physical registers of a processor. In a parallel processing environment, register file is shared among the processors which increases register pressure. To avoid the disadvantages of shared register files, each core in a multicore processors contains private register files. The compiler is responsible for allocating registers from the respective core for a program code or parallel regions of program which is scheduled to execute on that core.

The schedulers proposed in the chapter 6 & 7 creates schedules for cores in multicore processors. Each schedule contains sub-blocks. The register allocation for the instruction in the sub-blocks need to be done from private register file of each core.

The register demand of the sub-blocks can be either detected during scheduling (integrated approach) or after scheduling as shown in Figure 39.

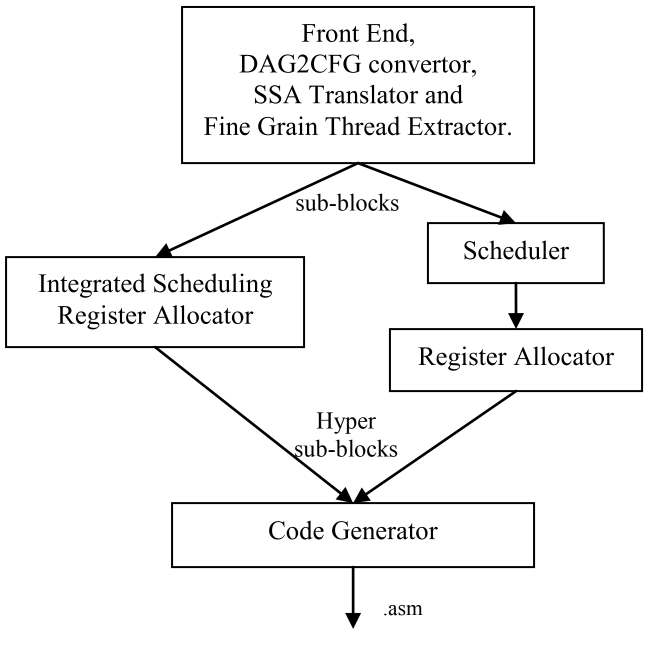


Figure 39. Proposed Register Allocation Heuristics

Four register allocation approaches are identified and investigated.

- A local register allocation heuristic (Heuristic 1) which follow Chaitin's approach.
- An integrated register allocation with global scheduling (Heuristic 2).

- A global register allocation heuristic (Heuristic 3) without simplifiability.
- A global register allocation heuristic (Heuristic 4) with simplifiability.

In Heuristic 2 register demand is also considered to make scheduling decisions. In heuristic 3 & 4 register allocation are performed after creating schedules for cores. The classification of these register allocation approaches are also based on region used to perform register allocation, i.e., whether the register allocation is done on individual sub-block or on hyper sub-block (merged sub-blocks). The Local register allocation approach uses sub-block and the global register allocation approach uses hyper sub-blocks formed by merging the sub-blocks in the schedule.

This chapter discusses the implementation details of the proposed register allocation heuristic 2, 3 and 4 for multicore processor. The significance of these heuristics are explained in Section 4.3 and implementation details are explained in section 8.1 and 8.2.

In general, the proposed register allocation heuristics creates a list of hyper sub-blocks H (h1,h2,h3....hx) whose interference graph is k-colorable as shown in Figure 18. Since the hyper sub-blocks are scheduled on a single core, it ensures temporal locality and reduces memory reference. The hyper sub-blocks which are k-colorable cause zero spilling and instructions remain inside private memory of individual cores till all the instruction commits without doing external memory reference.

8.1.Integrated Scheduling and Register Allocation (Heuristic 2)

This approach integrates register allocation pass with global scheduling [34]. The goal is to get away with the phase ordering problem and to overcome limitations which lead to poor optimizations. The scheduler schedules the sub-blocks in the dependency graph to multiple cores selectively, taking register requirement, dependencies and the order of execution into account.

The scheduler proposed here is a modified Dependent Sub-block based scheduler (DSB) [14], which keeps track of dependency between the sub-blocks in different basic blocks while scheduling onto the cores. The proposed work ensures that integrating register allocation phase with scheduling phase will not affect the performance of DSB scheduler.

The modified flow of the compiler after including fine grain extractor model to create disjoint sub-block and integrated scheduling phase is shown in Figure 40. The disjoint sub-blocks and sub-block dependency graph (SDG) are formed as discussed in section 5.1. The coloring module in the original compiler is removed and is integrated with the scheduling phase of the compiler.

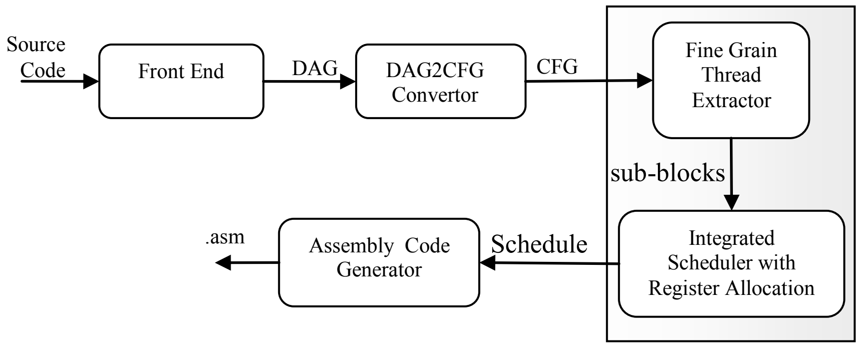


Figure 40. Modified Flow of Compiler To Integrate Scheduler and Register Allocator

Generally, the global scheduler selects the sub-block SB_iB_p from the sub-block dependency matrix if the dependency list of SB_iB_p is empty. Once SB_iB_p is scheduled and completes its execution, its entry is removed in all dependency lists. The decision of selecting sub-block are based on the invariants discussed in the section 4.2. The output of the scheduler is the list of sub-blocks to be scheduled on each core. The scheduler generates N schedules for a machine with N cores.

The integrated scheduler collects all the ready to execute sub-blocks to create ready list in non-incrementing order of the scheduling latencies of the sub-block (L_i). The register allocation algorithm is applied to check if the instructions in the sub-block are k-colorable. Live range of each operand in the instructions are computed locally in each sub-block. The scheduler creates the global interference graph incrementally by merging the live ranges of the sub-blocks while creating schedule for cores.

Initially the scheduler picks the sub-block with the highest schedule latency and schedules it on to any one of the core. Next scheduler picks the immediate ready successor of the scheduled sub-block in the SDG. If more than one immediate successor's are ready then, (a) The successor sub-block with the highest schedule latency is scheduled on to the same core. (b) If the schedule latency of successor sub-blocks are same, then the successor sub-block which results with minimum spill during register allocation is selected to schedule on the same core, while the other successor is scheduled on other core with least $T_{Stc.}$ If ready list is not empty and T_{Stc} of the current core is more than T_{Stc} of all the cores, the scheduler switches to the core with minimum T_{Stc}.

The sub-blocks which are in same schedule are merged to form the hyper sub-blocks if following conditions are satisfied.

- The merged sub-blocks will not affect the execution of the sub-block scheduled on other cores.
- The hyper sub-block is k-colorable.

Algorithm 14: Integrated Scheduling and Register Allocation

```
1. Find Latency
2. Sort sub-blocks by descending latency
3. Schedule & allot Registers:
  a. Single Core – in order of sorted list
  b. Multi Core:
      i.    temp ← top (list) (ready sub-block)
     ii.    Schedule temp & increment schedule time of this core
    iii.    Remove temp from list
     iv.    Update finish-time of temp and ready-time for all immediate successors
      v.    If any immediate successor of temp is ready (check in order of list) & list is non-empty
              If number of immediate successors > 1 and their latencies are equal
                  temp ← immediate successor which when scheduled results in colorable interference graph
              else
                  temp ← immediate successor not scheduled with highest latency
            goto step (ii.)
     vi.    If schedule time of current core is less than max schedule time & list is non-empty.
              goto step (i.)
    vii.    If the scheduled group is not colorable
                Spill to make interference graph colorable.
                Update schedule time of the core, finish-time of scheduled sub-blocks and ready-time for all the
                immediate successors of sub-blocks present in the scheduled group.
            else    Merge the sub-blocks in the schedule to create hyper sub-block by checking simplifiability.
   viii.    Max schedule time ← schedule time of current core
     ix.    If list is non-empty switch core goto step (i.)
      x.    END
```

The limitation of integrated scheduler is that, it works well when the sub-blocks exhibit high dependency in SDG. Dependency between sub-blocks facilitates creating larger hyper sub-blocks in the schedules. It is observed that integrating register allocation phase with other schedulers such as HIB, MDS and LLSF will not give desired gain in speed-up. As they increase the waiting time of other sub-blocks scheduled on other cores and increases the communication cost.

8.2. Global Register Allocation (Heuristics 4)

In this register allocation heuristic, the schedules generated by the scheduler is used. The algorithm incrementally merges the sub-blocks using Merge Operator to produces a list of hyper sub-blocks H (h1,h2,h3....hx) whose interference graph is k-colorable.

8.2.1 Merge Operator

The merge operator produces k-colorable hyper sub-blocks by merging the interference graphs of sub-blocks listed in schedule. While creating the hyper sub-blocks, the sub-block dependency and simplifiability conditions must be checked and satisfied. The algorithm for merge operator is given below. Algorithm begins by selecting two sub-blocks SB_i and SB_j which is followed by dependency constraint check. The constraints are enforced through the condition C1, C2 and C3 given below. These conditions are derived from the invariants used by the global scheduler (in Section 4.2).

Assuming that SB_j is listed in schedule of processor core Cr_a, the condition C1,C2 and C3 are checked to find if the sub-block SB_j can be merged with its predecessor sub-block SB_i to form hyper sub-block.

The sub-block SB_j can be merged with its predecessor SB_i iff it is not dependent on sub-block(s) SB_k where sub-block SB_k is scheduled on different core Cr_b where $a \neq b$. In case SB_j is dependent on sub-block SB_k it can be merged with its predecessor iff SB_k and SB_i have non overlapping execution i.e, finish time of SB_k is less than ready time of SB_i. The condition C1 and C2 are used for checking these two possibilities.

Condition C3 helps in reducing the wait time of sub-block SB_k. If a sub-block SB_k is scheduled on core Cr_b and is dependent on SB_i, merging of SB_i with its successors to form a hyper sub-block will cause SB_k to wait till the hyper sub-block execution is completed. To ensure zero spilling of the hyper sub-blocks, simplifiability condition C4 is checked. An example illustrating the merge operation is discussed in Section 4.3.

The conditions (C) and decisions (D) used in Algorithm 15 are given below.

I1: Let SB_j be the successor of SB_i in the schedule for core Cr_a.
Let SB_k be the sub-block in the schedule of other core Cr_b.

C1: If SB_j is dependent on SB_k.

C2: If T_{fns} of sub-block SB_k < T_{rdy} of SB_i.

C3: If SB_k is dependent on SB_i and T_{rdy} of SB_k is > T_{fns} of SB_j.

C4: If the interference graph of SB_i and SB_j are simplifiable and resulting interference graph after merging is also simplifiable.

D1: Merge the sub-blocks to schedule and allocate register together.

D2: Do not merge the sub-blocks.

Algorithm15 : Merge Operator

```
MergeOperation(sub-block $SB_i$ , sub-block $SB_j$)
begin
   initialize
     $SB_j$ be the successor of $SB_i$ in the schedule for core $Cr_a$.
     $SB_k$ be the sub-block in the schedule of other core $Cr_b$.
    if(C1 & C2 & C4)
    begin
       Merge the sub-blocks to schedule and allocate register.

    End if
    else if(!C1)
    begin
       if(C3 & C4)
       begin
          Merge the sub-blocks to schedule and allocate register.
       End if
    End else if

    else
    begin
       Do not merge the sub-blocks (D1).
    End else
End
```

The disjoint-set forests [107] algorithm can be used for merging the interference graph. The union-by-rank heuristic is used to improve the runtime of union operation and path-compression is used to improve the runtime of the find set operation.

8.2.2 Observation on Number of Registers

This section discusses the effectiveness of proposed register allocation when the number of registers are varied. The results shown in the section 8.4 is for the cores having 8 general purpose register each. The effect of increasing the number of registers leads to reduced spilling when heuristics 1, 2 and 3 are used which is obvious.

As interference graphs are built incrementally by checking dependency and simplifiablity conditions. Increasing the number of registers can cause increased number of instructions in the hyper sub-block resulting into optimized code generation requiring lesser execution time. The improved execution time can be attributed to the fact that larger hyper sub-block will require less data movement to and from memory.

According to chromatic polynomial theory for the Chordal graph, a fully connected Chordal graph with k nodes need k colors. The interference graph of the SSA form program will never be fully connected and most of the time it is 3 colorable.

The interference graph of the benchmark programs used in this work shows either of the following coloring pattern for $k+1 \leq 8$ where k range from 3 to 7.

- sub-blocks are k colorable and hyper sub-block is also k colorable.
- sub-blocks are k colorable and hyper sub-block is k+1 colorable.

There will be no change in performance if number of registers used is reduced to 3. Further, the performance will not change if more than 8 registers are used, because of dependencies between the sub-blocks scheduled on different cores.

8.2.3 Capturing Live Variable

The Live variables are captured during SDG creation. In the SDG, the sub-blocks are represented as vertices V and dependency between the sub-blocks are represented by directed edges E. The total number of Live_in variables of the sub-block SB_jB_q is the degree of dependency between SB_jB_q and SB_iB_p, i.e. total number of variables involved in the dependency. Live variables in a sub-block SB_jB_q is the sum of degree of dependency of all incoming edges and variables that are defined in the sub-block.

8.2.4 Register Assignment

In this phase, the live variables in the hyper sub-blocks are assigned register. As the simplifiability condition is checked during the formation of hyper sub-block, the need to insert spill code is eliminated. The choice of the order of coloring is simplified due to the fact that the interference graph is Chordal with simplical vertex. The edge projecting out of the simplical vertex is pushed on to the color stack first and continued till all the edges are pushed on to the stack. Once all the edges are pushed on to the stack, the color assignment module pops out the edges from the stack to assign different color for the conflicting edges. The color stack is used to prioritize the coloring i.e the edge in higher position in the stack is given higher priority.

8.2.5 Insert Spill Code

In this phase, the spill code load/store is inserted for the spilled variable which are captured during construction of initial interference graph of the sub-blocks. However the interference graphs of the hyper sub-blocks are k-colorable which eliminates the need of spill code. The spill codes are inserted after creation of hyper sub-blocks and register assignment phase to retain the properties of SSA.

8.3. Algorithm Complexity

In this section complexity analysis for the proposed register allocation heuristic is presented. The complexity of the other three heuristics are compared and presented in Table XI.

Let m and n be the number of sub-blocks and number of live-ranges or variables. The time consumed per sub-block to check the presence of dependent sub-blocks or parental sub-blocks on other cores is O(m). The mergeSubblock module has O(n log n) complexity. Thus, for heuristic 3 the complexity of all the iterations (for m sub-blocks) is O(m*nlogn). In heuristic 4, the time consumed to verify the individual interference graphs for simplifiability, merging of the two interference graphs, and verifying if the new graph is simplifiable is $O(n^2)$. The complexity of coloring module is O(n log n). Thus, the overall complexity of the checkSimplifiable and mergeSubblocks modules is the order of $O(n^2 + n \log n)$ i.e., $O(n^2)$. Thus, the complexity of all the iterations (for m sub-blocks) is $O(m*n^2)$.

TABLE XI. ALGORITHM COMPLEXITY COMPARISON OF DIFFERENT REGISTER ALLOCATION HEURISTICS

	Heuristic 1	Heuristic 2	Heuristic 3	Heuristic 4
Complexity	O(n*log(n))	$O(m^2 * n^2)$	O(m *nlog(n))	$O(m* n^2)$

At first sight, it might appear that heuristic 4 has big increase in compilation time. But this complexity is acceptable, when overall compilation process is considered, as the complete code generator including register allocation contributes less than 20% of the total compilation time.

8.4. Results

The amount of spill caused by four different register allocation heuristics is computed and its effect on speed-up, power consumption and performance per power are compared.

- Heuristic 1 does not contribute spill code as the interference graph of the sub-blocks are k-colorable. This heuristic requires the runtime environment to assign threads to individual sub-blocks and handles corresponding frequent data movement from the memory.
- Heuristic 2 tries to solve the phase ordering problem by allocating register during scheduling by compromising a little with the performance. This heuristic results in reasonable amount of spill code as well as increases the compilation time.

- The heuristic 3 overcomes the problem faced with heuristic 1 by allocating threads to hyper sub-blocks instead of sub-blocks. But this heuristic results in greater amount of spill code.
- Heuristic 4 by checking the simplifiablity conditions for the hyper sub-block, combines the feature of heuristic 1 and 3 resulting into spill code elimination and reduced runtime environment overhead.
- The amount of spilling when the four heuristics are used is shown in Table XII. The spilling is almost zero when register allocation is done on the list of sub-blocks using heuristic 1 as sub-blocks are created by taking register requirement. Similarly spill is zero in the proposed heuristic 4 as interference graph of hyper sub-blocks are k-colorable.

TABLE XII. SPILL COMPARISON OF DIFFERENT REGISTER ALLOCATION HEURISTICS

Algorithm		Heuristics 1	Heuristic 2	Heuristic 3	Heuristics 4
Test Case 1	Dual Core	0	1	0	0
	Quad Core	0	1	1	0
	3- Active Cores	0	#######	1	0
Test Case 2	Dual Core	0	1	3	0
	Quad Core	0	1	0	0
	3- Active Cores	0	#######	0	0
Test Case 3	Dual Core	0	0	3	0
	Quad Core	0	0	0	0
	3- Active Cores	0	#######	0	0
Test Case 4	Dual Core	0	1	2	0
	Quad Core	0	0	2	0
	3- Active Cores	0	#######	2	0

TABLE XIII. COMMUNICATION COST OF INTRA BLOCK, INTER BLOCK AND INTEGRATED SCHEDULER

Test case No	Communication Cost		
	Intra block Scheduling	Inter block Scheduling	Integrated Scheduling
T1 – Dual	13	7	5
T2 – Dual	19	10	8
T3 – Dual	11	0	0
T4 – Dual	10	4	4
T1 – Quad	15	10	9
T2 – Quad	26	18	22
T3 – Quad	9	0	0
T4 – Quad	22	4	4

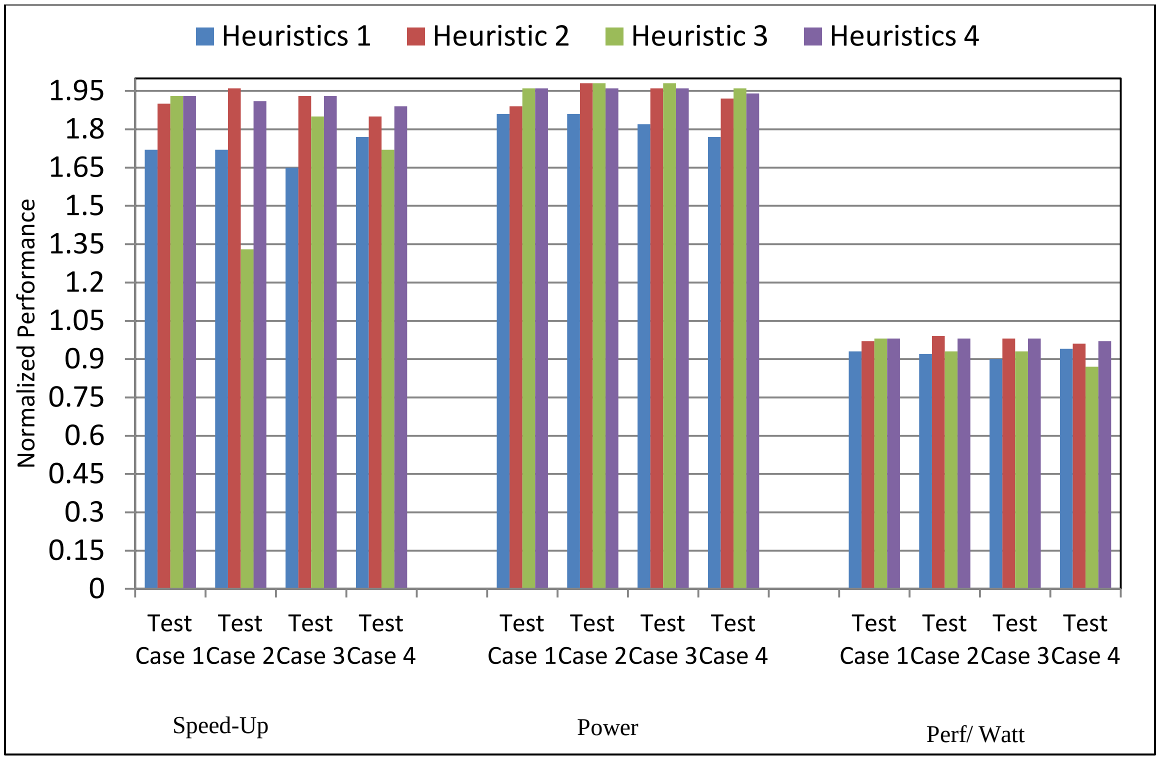


Figure 41. Speed-up, Power and Perf/Power Comparison on Dual Core Machine

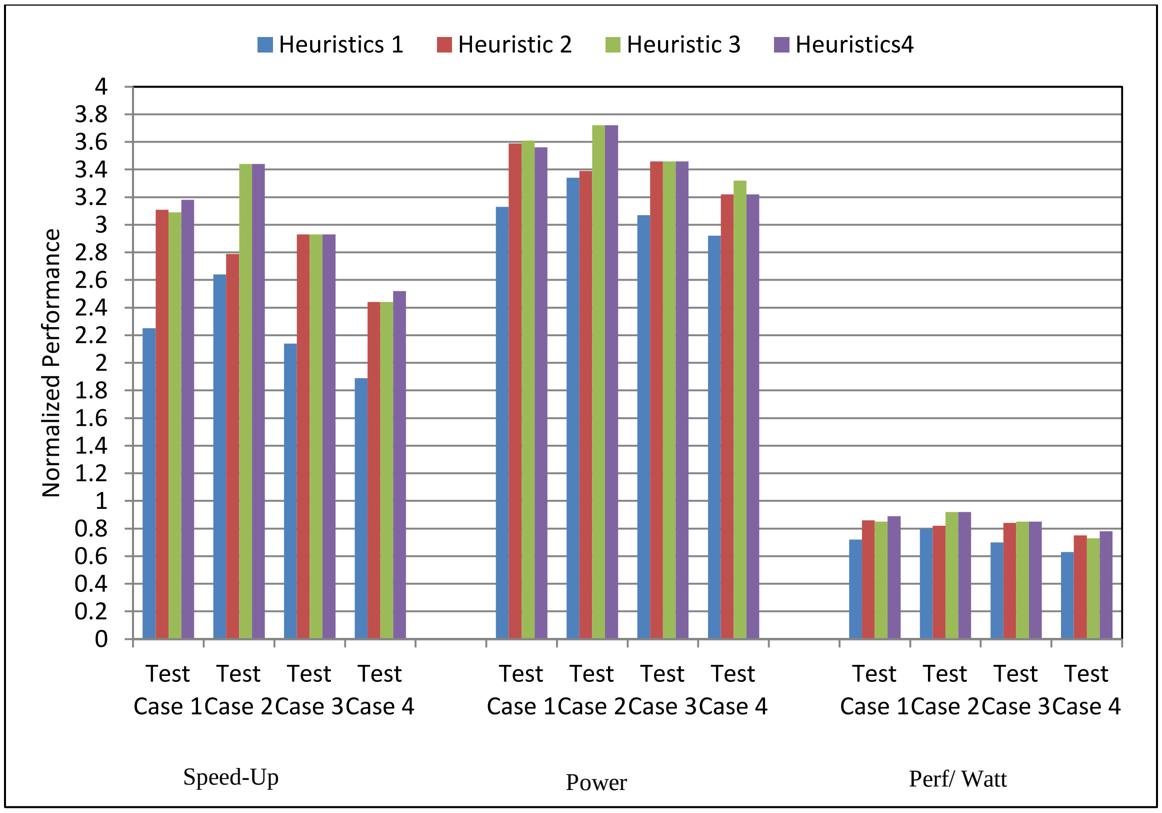


Figure 42. Speed-up, Power and Perf/Power Comparison on Quad Core Machine

The performance gain is a combined effort of scheduler and register allocation approach. The results in Figure 41 and 42 depict the effect of spilling on speed up, power and Perf/watt on dual core and quad core processor.

- The speed-up and performance per watt of heuristic 1 is lower even with zero spills. This is because of limitation of the local scheduler. The local scheduler assigns thread to each sub-block resulting in higher data movement.
- Heuristic 2 shows better speed-up for Test Case 2, this is because the integrated scheduler is able to create a schedule for dual core processor efficiently. The performance of the heuristic 2 is deteriorated for test case 2 on quad core processor due to memory contention.
- Speed up-decreases when spill increases, due to insertion of extra spill instructions. When heuristic 3 is applied on Test Case 2, 3 and 4 to execute on dual core machine, the speed-up decreases due to spilling. This results in higher power consumption and reduced performance per watt in comparison to heuristic 4.
- Test case 3 on quad core processor shows same performance on all the heuristics in terms of speed up as there is no spill in either of the heuristics.
- Heuristic 4 performs better for all the test case on quad core processor.
- It is clear from Figure 41 and 42 that the overall performance of heuristic 4 is better with respect to performance per watt.
- Heuristic 4 tries to achieve better performance with improved compilation time.

8.5. Conclusion

The work proposes a register allocation mechanism to be used for multicore processors. The proposed mechanism promises to give better results than those obtained using a conventional register allocation mechanism built for single core processors. The experimental results presented in this chapter endorse this fact. The algorithm takes into account the presence of multiple cores and the presence of their separate register files, and exploits this avenue to achieve better register allocation results. Four heuristics for allocating registers for fine grained threads are discussed. The spills, speed-up, power consumption and performance per power are compared.

CHAPTER 9 *Achievements, Limitations, Future Work, and Summary*

In this chapter, we conclude our research work by summarizing the achievements, providing limitations and suggest directions for future research.

9.1 Summary of Achievements

i. The sub-blocks are created from sequential program by checking the true dependencies between the instructions. The instructions inside the sub-blocks are true dependent on each other which ensures the spatial locality.

ii. The Local and Global schedulers are proposed to achieve high speed-up and low power consumption.

iii. Cache coherence is a major concern in multicore environment where L2 cache is shared. Since sub-blocks in a basic block of CFG are disjoint, it solves the problem of cache coherence when scheduled locally on multiple cores. However indirect cache coherence exists which our method will not identify.

iv. Schedulers also provide the solution to memory contention as the dependent sub-blocks are not scheduled together at same time.

   iii and iv mitigate the runtime overheads like, locking, synchronizing....etc.

v. To ensure temporal locality, all the data belonging to sub-block are moved on to Instruction and Data cache/memory of the core on which sub-block is scheduled and remain their till it commits.

vi. To reduce data movement between core and memory as sub-blocks are merged to create Hyper sub-blocks. Hyper sub-blocks ensure more instructions provided for execution in one fetch.

   Sub-blocks are also merged for the other two reasons

   - To perform power optimization.
   - To ensure efficient register allocation.

vii. The schedulers are designed to perform power optimization i.e., if the speed-up achieved using n cores can be achieved using n-1 cores, then only n-1 cores of the n core machine are used to execute the given task, either by keeping the n^th core idle or utilizing it for some other computation.

viii. When the sub-blocks are scheduled individually on to the multiple cores, we see that number of registers required by the sub-block is less than or equal to number of register in each core. If number of register required is less than the registers available, scheduler tries to combine the sub-blocks whose register requirement is almost equal to register available in the core. This ensures there is no register spill during execution when individual sub-blocks are scheduled.

ix. Hyper sub-blocks are created before register allocation to ensure temporal locality by pushing maximum instructions on to core for execution. Hyper sub-blocks also ensures that instructions will do zero spills (k-colorable) and will remain in cores private memory till it commit without doing memory reference during execution.

9.2 Limitations

- The idea of creating disjoint sub-blocks and creating schedules for multicore processor is successfully implemented on Jackcc compiler. Though the Jackcc compiler is capable of compiling C program, the experiments are done using limited set of instructions and operators. Due to this limitation of Jackcc, the scalability test for the proposed schedulers were not performed for different Test Cases. Our results shows that our schedulers scale on dual and quad cores but did not scale well on 8 and 16 cores as the scalability of our technique is directly proportional to the amount of inherent ILP within the target program.

  The scalability test is performed by creating schedule for multiple instances of the test cases. The speed-up achieved are shown in **Appendix A**

- The explicit parallel programming techniques and compiler to compile these explicit parallel programs has reached a different height and are able to scale for the processors with accelerators. This is because, researchers are concentrating on developing programmer friendly technique to write parallel program. The work proposed in this thesis is an alternative and a complimentary technique where programmers need not write parallel program, instead depend on compiler to extract parallelism in the sequential program. This work is just an initiative and lot of work need to be performed to catch the speed to multicore design.

9.3 Future Work

- The future work involves porting the proposed techniques onto LLVM or GCC compiler and to make it open source so that hackers can contribute to improve the technique.
- To extract fine grained threads that can be scheduled on multicore processors with multiple functional units and to allocate register based on type operand and frequency of its use.
- To profile a sequential program to pre-fetch the data required by fine grained threads to facilitate parallel access of the data.
- To design the inter procedural scheduling.
- To analyze the scheduling algorithm specific to memory contention.
- To develop a STM to monitor the schedule created by the compiler.

9.4 Summary

Multicore has emerged as the mainstream processor design paradigm in the field of computer architecture. For most of the existing applications the performance is not directly translated. The existing application need to rewritten using explicitly parallel programming techniques or need to depend on tools such as compilers to parallelize the sequential code and runtime environment execute utilizing all computational capabilities in multicore environment.

The proposed research provides compiler support to exploit parallelism by extracting fine grained threads from a sequential program. The fine grained threads are heuristically scheduled on multiple cores to achieve speed-up by effectively utilizing computing capability of multicore environment. The fine grained threads that are scheduled on to different cores are allocated registers from respective register file of the core on which they are scheduled.

Thesis starts with a brief introduction to design philosophy and challenges in multicore architecture. Several dynamic and static support to exploit ILP in the existing parallel architectures such as Pipeline, VLIW, Superscalar and Multi clustered VLIW architectures are investigated as part of literature study. An effort is made to understand the pros & cons of hardware and compiler approaches. The challenges in multicore environment, motivations and a brief description of the steps involved in the proposed work to face the challenge is given.

The work proposes various compiler level scheduling techniques for multicore processors. The goal underlying these techniques is to promote exploiting of ILP in multicore environment without explicitly specifying parallelizable fraction of the program by the programmer. To achieve it, the basic blocks of the control flow graph of a program are formulated into multiple sub-blocks. The compilation time efficient technique to form disjoint sub-blocks using two different approaches are proposed. To facilitate global scheduling new data structure called sub-block dependency graph (SDG) is proposed and efficient technique to create it is discussed in detail.

The scheduler's prepares the schedule for multiple cores selectively, taking the dependencies among the sub-blocks into account to maintain the order of execution. The local scheduling heuristics (Intra Block Scheduling) which schedules the parallel regions within the basic blocks of CFG is discussed in Chapter 06 and four global scheduling heuristics (Inter Block Scheduling) which schedules the parallel regions formed across the basic block of the CFG is discussed in chapter 07. The brief discussion on merits and demerits of each heuristics are presented by comparing the results obtained by them. The results obtained by Intra block scheduling is compared with the results obtained by Inter block scheduling technique. These schedulers also carry out locality optimizations to minimize communication latency among the cores and to minimize the overhead of hardware based instruction reordering.

A detailed survey on register allocation approaches are presented in chapter 2 and register allocation technique for multicore architecture is presented in chapter 8. The proposed mechanism, which has been tailor made for use on a multicore processor, promises to give better results than those obtained using a conventional register allocation mechanism built for single core processors. The experimental results presented endorse this fact. To mitigate the phase order problem a new approach to integrate register allocation with global scheduler is presented. The goal of underlying technique is to overcome limitations which lead to poor optimizations and had bad impact on ILP. The proposed scheduler creates schedule for the cores by taking register requirement of sub-blocks and dependencies among the sub-block into consideration. The results obtained by the normal register allocation approach and integrated approach is compared and presented in the end of the chapter 08. The efficiency of register allocation technique is measured in terms of spills done by four different heuristics and their effect on speed-up, power and performance per power is also shown.

The code generated for the programs in RAW benchmark suite is analyzed and compared for outcome of inter block and intra block scheduler. The performance analysis is based on the metrics such as speed-up, power consumption, performance per power and inter-core communication latency.

The schedulers also do locality optimizations to minimize communication latency among the cores. Though the results are shown on dual core and quad core processors, the proposed schedulers are scalable to any number of cores which are available in the modern architectures. The results which illustrate the scalability is given in **Appendix A**.

APPENDIX A *More Results on Scalability*

Scalability of the proposed work is analyzed by creating schedule for multiple instances of test cases for 8 core and 16 core processor. The Figure 43, 44, 45 and 46 illustrate the speed-up achieved and observations are listed as follows.

- Executing multiple instances on Dual core and Quad core processor does not change the speed-up as it is proportional to time and number of instruction getting executed in that time. Similar type of observation can be made on power and performance per power by following Woo-Lee model.
- The single instance of test cases does not scale much on 8 and 16 core processors.
- Two instance of all four test cases scale well on 8 cores.
- Speed-up decreases when three instances of all four test cases are executed on 8 cores. This is because the third instance adds extra execution time. Similar observation can be done on executing four instances on the eight core processor.
- Speed-up is gained when 3 instances and 4 instances of all four test cases are executed on 16 cores.
- It can also be observed that, the proposed schedulers can perform active power optimization. For example, if three instance of application or program is compiled for 16 core processor, the speed-up achieved may not be so pleasing compared to executing it on 12 cores. In such case, the compiler creates schedule for 12 cores instead of 16 cores keeping 4 cores idle or allowing operating system or STM to schedule other application on these 4 cores. Figure 47 and Figure 48 illustrate this fact. The schedule for four instances of test case 1 is created for 12,13,14,15 and 16 cores and the schedule which requires minimum number of core to achieve maximum speed up is selected.

With these observations, it can be concluded that the proposed work scale well with number of cores if the application or program exhibit parallelism. The fine grain thread extractor module exploit as much parallelism compared to explicit parallel program written by programmer.

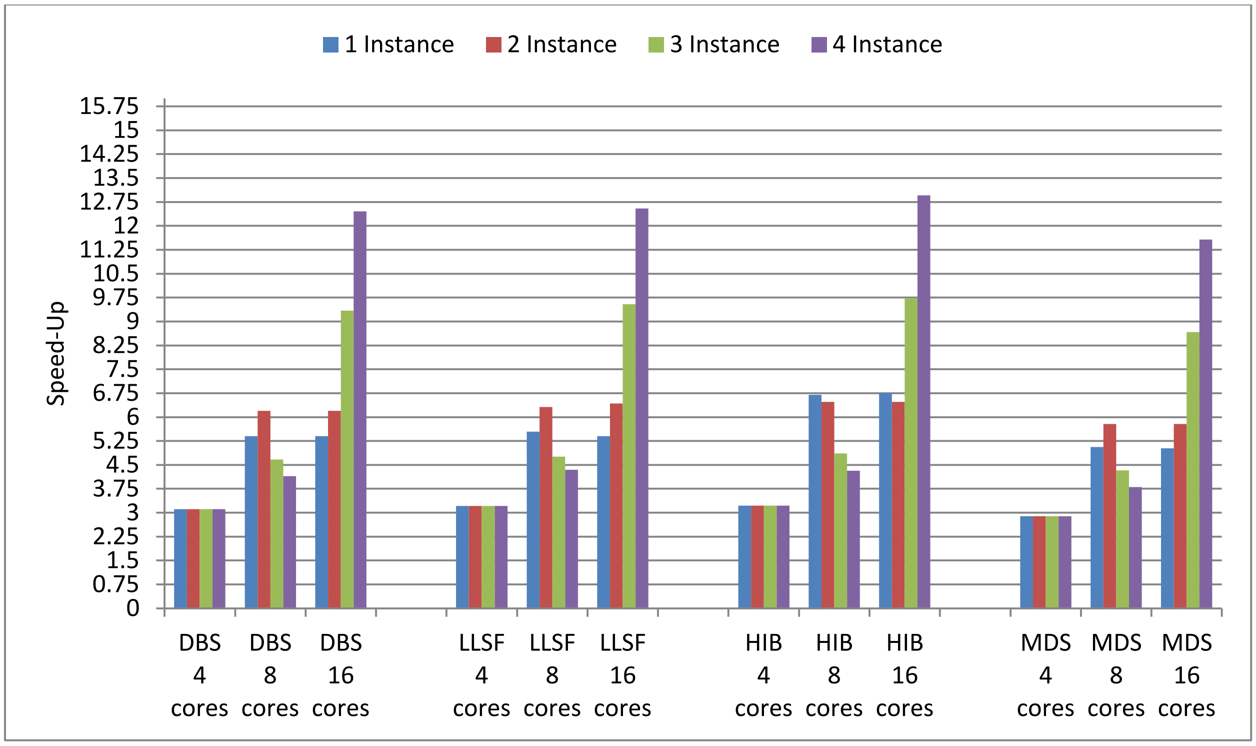


Figure 43. Speed-up Achieved on 4, 8 and 16 cores by Executing Multiple Instances of Test Case 1

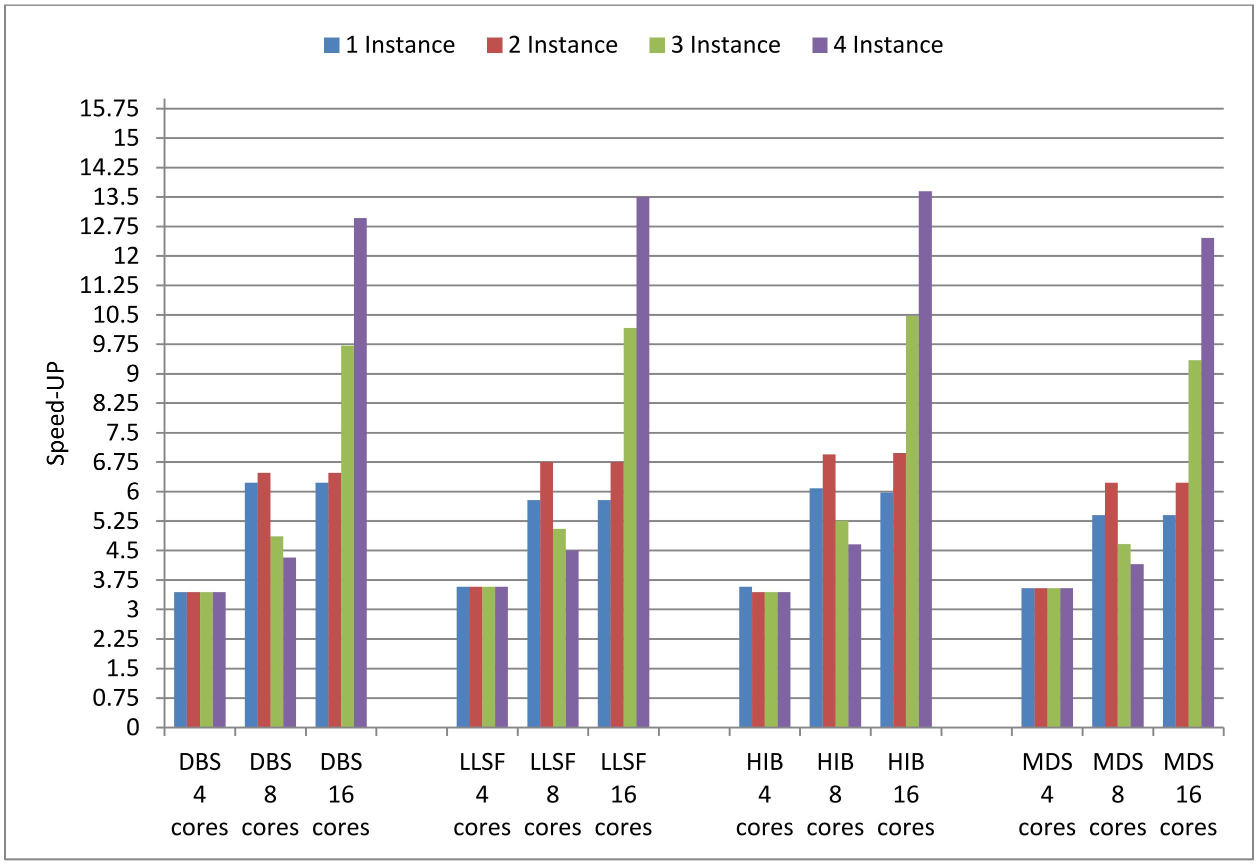


Figure 44. Speed-up Achieved on 4, 8 and 16 cores by Executing Multiple Instances of Test Case 2

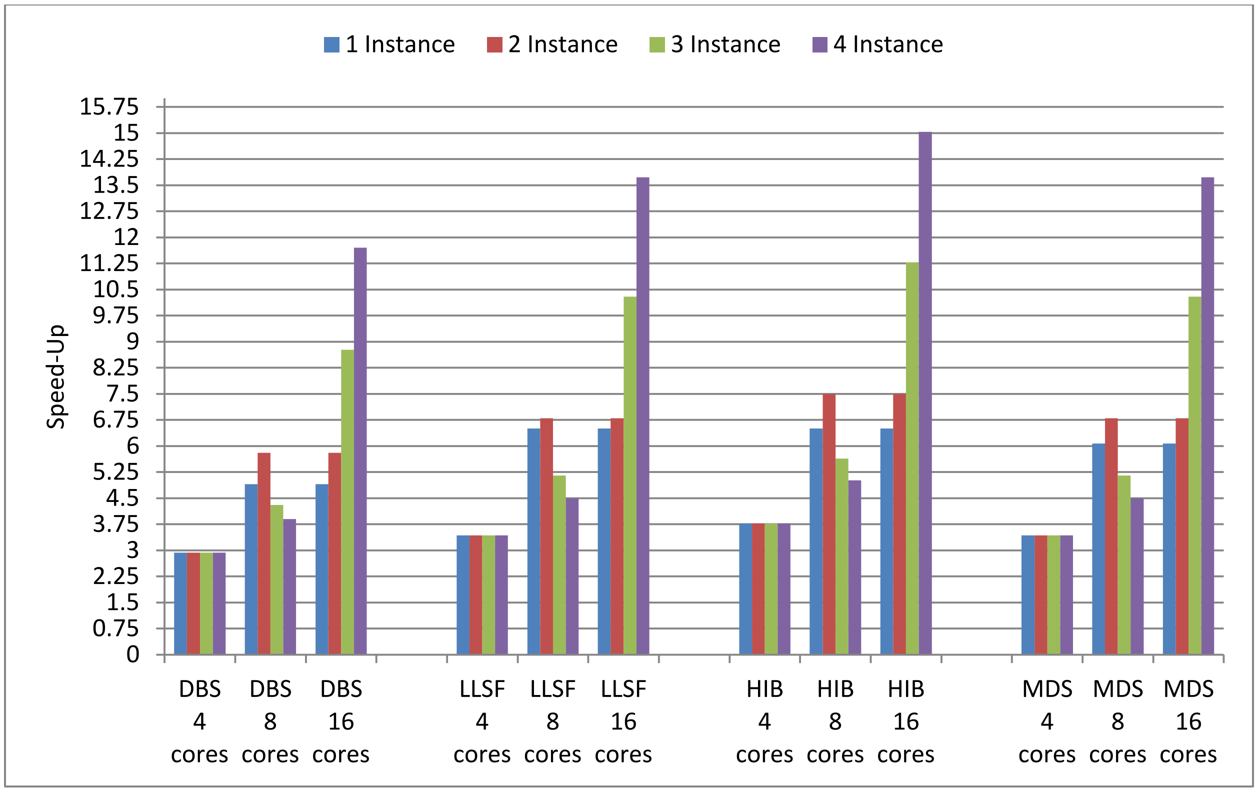


Figure 45. Speed-up on 4, 8 and 16 cores Acheived Executing Multiple Instances of Test Case 3

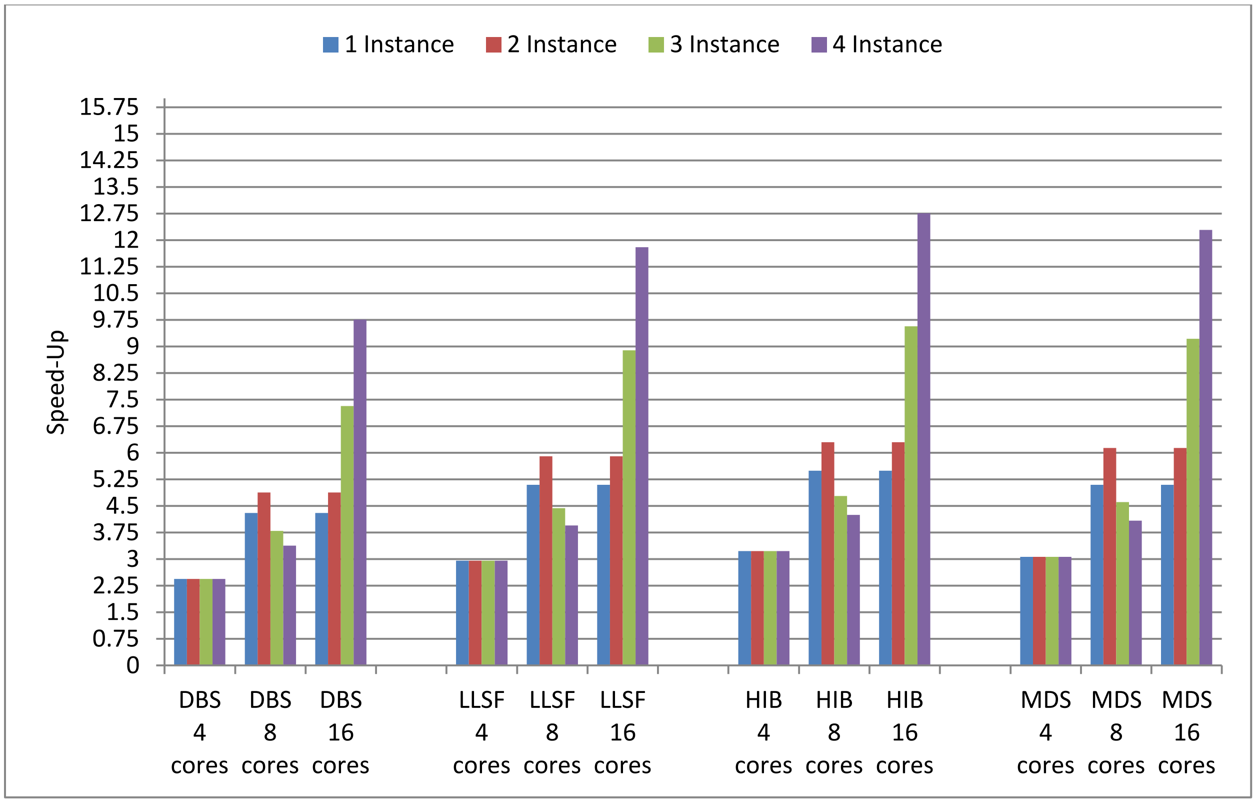


Figure 46. Speed-up on 4, 8 and 16 cores Acheived Executing Multiple Instances of Test Case 4

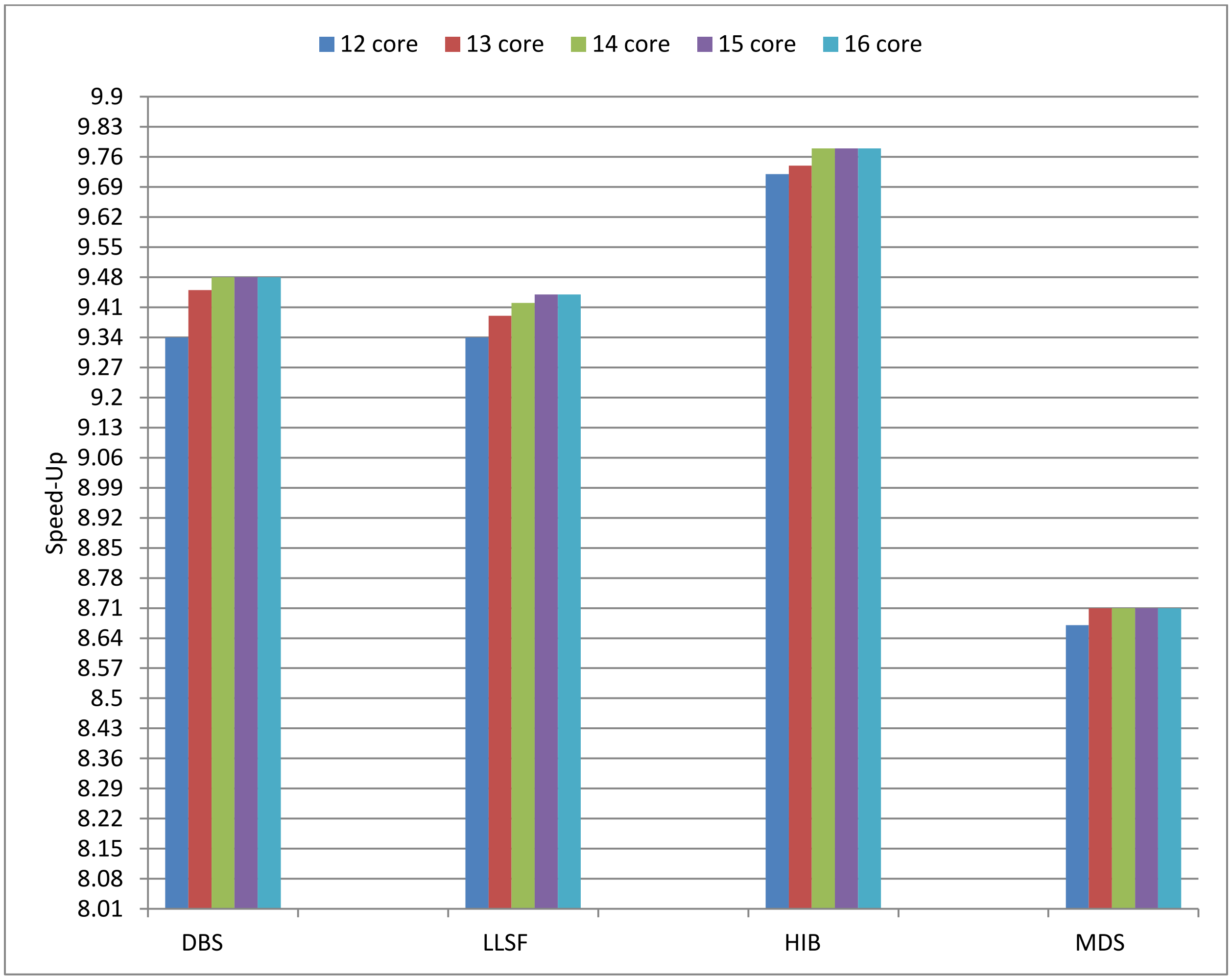


Figure 47. Speed-up Analysis by Executing 3 Instances of Test Case 1 on 12, 13, 14, 15 and 16 core for Power Optimization

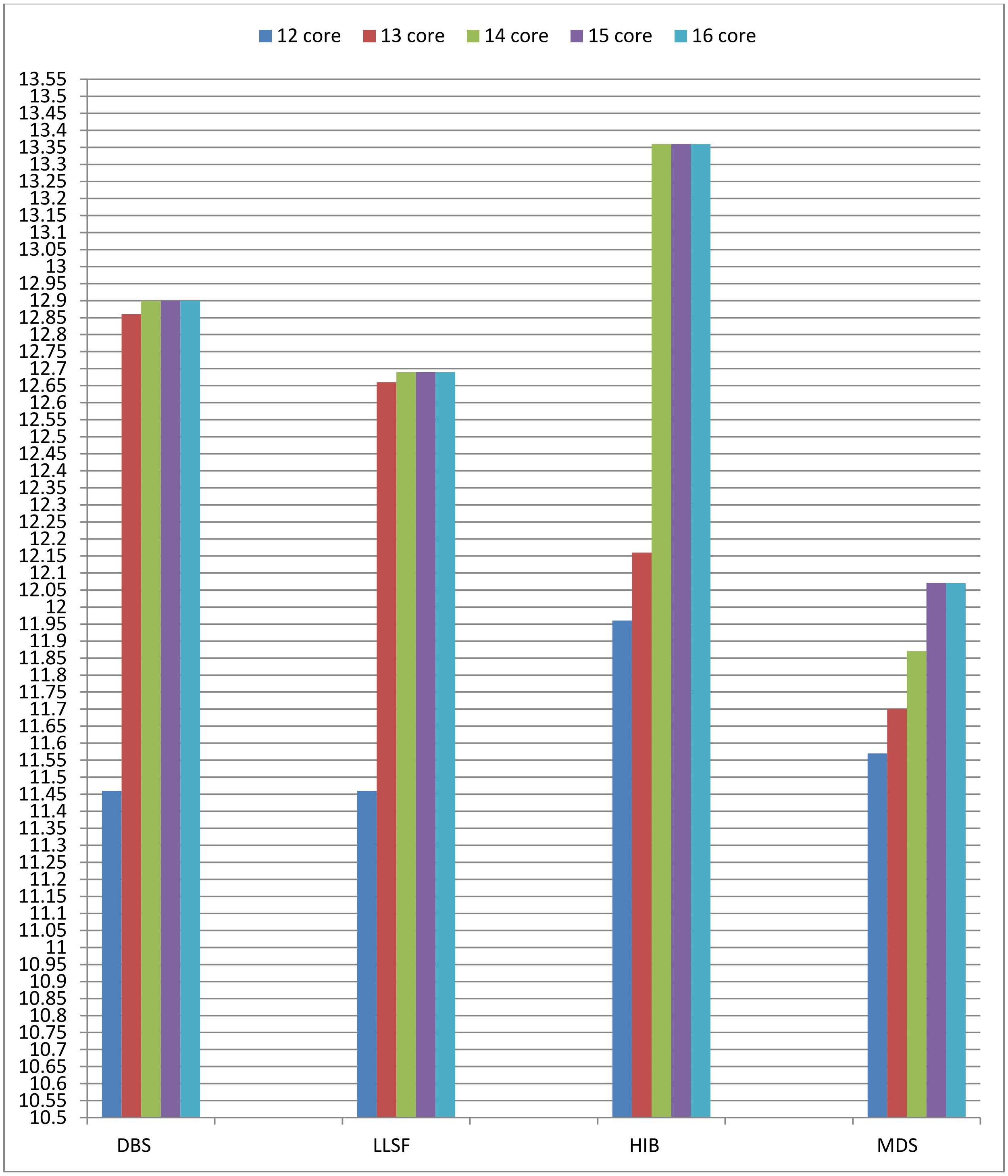


Figure 48. Speed-up Analysis by Executing 4 Instances of Test Case 1 on 12, 13, 14, 15 and 16 core for Power Optimization

APPENDIX B *Preliminaries and Definitions of Jackcc Compiler*

This section is to provide the details of Jackcc compiler. The Jackcc compiler is an Optimizing C Compiler. Jackcc generate assembly code using the Jackal 3.0 ISA, whose instructions are listed below. Jackcc is developed by Nick Johnson to use in the University of Virginia. Additionally, this project has developed an assembler and simulator, Jackas for the Jackal 3.0 ISA.

The Jackcc is written in C language. The different compilation phases and optimization pass are designed and implemented as set of reusable libraries. The Table XIV provide the list of files of Jackcc compiler. The flow of compiler and detail description are provided in Figure 6 and section 3.2 of Chapter 3. The supporting files are listed in Table XV (interfaces with .h extension).

TABLE XIV. FILES IN JACKCC

arith.c	jump.c	symtab.c	lex.yy.c
color.c	scav.c	toofar.c	dist-lex.yy.c
cse.c	loops.c	util.c	dist-y.tab.c
dag.c	parser.y	vars.c	scanner.l
dag2quad.c	peep.c	t.tab.c	ytab.c
ioc.c	**quads.c**	fcn.c	
parms.c	semantic.c		

TABLE XV. SUPPORTING FILES IN JACKCC

arith.h	jump.h	symtab.h	lex.yy.h
color.h	scav.h	toofar.h	dist-lex.yy.h
cse.h	loops.h	util.h	dist-y.tab.h
dag.h	parser.y	vars.h	scanner.l
dag2quad.h	peep.h	t.tab.h	ytab.h
ioc.h	quads.h	fcn.h	
parms.h	semantic.h		

The ADT's defined in Jackcc are listed in Table XVI. The corresponding files containing these definitions are also provided in the table. The type whose names prefixed with *s* are defined using structure of C language. Similarly, the names prefixed with *e* and *u* are defined using enumeration and union of C language.

TABLE XVI. ABSTRACT DATA TYPES DEFINED IN JACKCC AND CORRESPONDING FILES

Structure Name	**Data type**	**Containing File**
s_live	Live	color.h
s_graph	Graph	color.h
s_quad	Quad	quads.h
s_arena	Arena	quads.h
s_symbol	Symbol	symtab.h
s_dag	Dag	dag.h
s_annot	DataAnnotation	dag.h
s_stack	Stack	util.h
s_bst	Bst	util.h

e_quad type	QuadType	Quads.h
e_live type	LiveType	Color.h
e_symtype	Symtype	Symtab.h
e_dagtpe	DatType	Dag.h
u_dual	Dual	util.h

B1. Instructions Supported and Their Meaning

Instructions used to create assembly code (ISA) is provided in Table XVII. The dest, sa, sb are all registers.(sa) means the memory pointed to by the memory address in register sa. Instructions are shown on the left and their meanings on the right.

TABLE XVII. INSTRUCTIONS USED TO GENERATE ASSEMBLY CODE BY JACKCC

Instruction	**Meaning**
ADD dest, sa, sb	dest←sa +sb
SUB dest, sa, sb	dest←sa - sb
MUL dest, sa, sb	dest←sa * sb
AND dest, sa, sb	dest←sa && sb
OR dest, sa, sb	dest←sa || sb
NAND dest, sa, sb	dest←sa NAND sb
SLA dest, sa, imm	dest←sa<<#imm
SRA dest, sa, imm	dest←sa<<#imm

LD dest, sa	dest← (sa)
ST sa, sb	(sa)←sb
CONST dest, imm	dest←#imm
OFFSET dest, imm, sym	dest←#imm + OFFSET of label(sym->name)
CMP sa, sb	compare register contents of sa and sb
JPOS sym	jump id positive flag is set to label sym->name
JNEG sym	jump id positive flag is set to label sym->name
JZERO sym	jump id positive flag is set to label sym->name
JUMP sym	jump to label sym->name
JREG sa	jump to address in register sa
MOVE dest, sa	dest←sa
LABEL sym	this converts to assembly as a label statement LABEL sym->name
SAME	For internal purpose. To specify to instruction are copy instruction. (Used to represent SSA form programs)
LINE	Used for internal purpose
TOUCH	Used for internal purpose

B2. Basic Abstract Data Types

B3.1. Stack type.

```
union u_dual
{
        int     ival;
        void    *pval;
};
typedef union u_dual    Dual;
```

```
struct s_stack

{

Dual value;

struct s_stack *next;

};

typedef struct s_stack    Stack;
```

B3.2. Binary Search Tree Type.

This forms the basic block of the Tree, it can either contain an integer value, or a void pointer.

```
struct s_bst
{
        Dual    value;
        struct s_bst     *  left,  *  right;
};

typedef struct s_bst     Bst;

typedef Dual (*BstCallBack)(Dual, Dual);
```

This is a function pointer which can call a function with two DUAL arguments and return a DUAL value.

B3.3. Triangular Bit Matrix data type

```
typedef  int Tbm
```

B3.4. Symbol Table:

This is to store what is the type of the value stored in the symbol table.

```
struct s_symbol
{
        int   id;

        SymType type; //  The type that is defined in the structure above

        char *name; // The particular name of the symbol table entry

        int offset;
         //The offset of the record from the particular start point, could be the frame pointer position
        int   value;        //The integer value, if it has an integer value

        struct s_symbol    *   mode;

       int          size;     // the number of entries that are there in the table, the cardinality.

        int         flat_uses;

       int          weighted_uses; //count for number of times used.

     struct s_symbol       *        parent; //Pointer to the structure, or function or block of code which
contains this symbol, for back referencing

        // The following fields are only used for functions

        Bst       *called_by;

        Bst       *returns_to;

        Bst       *call_to;

        struct s_symbol       *  structure_fields;
        //Head pointer to the list of fields in the function. I don't think this is the most optimized way of
        keeping the fields

        struct s_symbol       *     function_formals; //Head pointer to the linked list of formal parameters

       struct s_symbol      *       function_locals;             //Head Pointer to the list of local parameters

       int          can_put_in_register;      //Can this value be stored in a register or not

      int           constant_in_register;

      int           dirty;    //To reduce the number of caller saves for this particular variable.
};
typedef struct s_symbol              Symbol;
```

B3.5. Symbol Type:

```
enum  e_symtype
{
          // an unused entry                              ST_UNUSED=0,
          // a global variable                            ST_GLOBAL,

          // a formal parameter                           ST_FORMAL,

          // a local variable                             ST_LOCAL,

          // a structure definition                       ST_STRUCTURE,

          // a union definition                           ST_UNION,

          // a field in a structure or union              ST_FIELD,

          // integer type                                 ST_INTEGER,

          // array-of type                                ST_ARRAY,

          // a function                                   ST_FUNCTION,

          // a function prototype                         ST_PROTOTYPE,

          // a constant                                   ST_CONSTANT,

          // a label                                      ST_LABEL

};

typedef enum e_symtype          SymType;
```

B3.6. Quad Type (for Instruction):

```
enum e_quadtype
{
          // declare a label at this position                     QT_LABEL=0,
          // make note of line numbes                             QT_LINE,
          // do nothing but affect register colorer
          // i.e. pretends to read a register                     QT_TOUCH,
          // affect register colorer -- demand that its
         // two args occupy the same register.                    QT_SAME,
        // arithmetic operations                                  QT_ADD,
                                                                  QT_SUB,
                                                                  QT_AND,
                                                                  QT_OR,
                                                                  QT_NAND,
                                                                  QT_SLA,
                                                                  QT_SRA,
                                                                  QT_MOVE,
         // memory access                                                QT_LD,
                                                                  QT_ST,
 // constants                                                     QT_CONST,
                                                                  QT_OFFSET,
// control flow                                                   QT_CMP,
                                                                  QT_JPOS,
                                                                  QT_JNEG,
                                                                  QT_JZERO,
                                                                  QT_JUMP,
// two optional instructions
                                                                  QT_MUL,
                                                                  QT_JREG
};

typedef enum e_quadtype      QuadType;
//It explains what basically the Quad accomplishes.
```

```
struct s_quad
{
      QuadType      type;    //What is the function of this Quad.
      int           dest;
      int           sa, sb;
      Symbol * sym;
      int           imm;     // the loop depth of this quad
      int     loop_weight;
      struct s_quad    *next,* prev;
};

typedef struct s_quad    Quad;
```

B3.7. Basic Block Type:

```
struct s_arena
{
      Quad    *first, * last;   //Pointers  to the first and last Quad in this particular Arena

      int     num_quads;     //Count for the total number of Quads in this arena
      int     code_unreachable;    //Used when appending to the arena, so that unreachable code can
                                   automatically be removed
      int     last_line;
};

typedef struct s_arena    Arena;
 //An Arena is a collection of Quads, helping us to describe  a Block.
```

B3.8. **DAG – Data Structure**

A lot of optimizations can easily be done on the graph, such as constant folding, common sub expression elimination, and reduction in strength.

Each node has an associated mode type (mode), and optionally a pointer to a symbol table entry. Each node has a unique id number too, of course.

```
struct s_annot
{
    // we have 32-booleans that we can use.
        int     flags;    // the depth of this instruction in loops
        int     loop_weight;   // the line of code that generated this  instruction
        int     source_line;
};
typedef struct s_annot   DagAnnotation;
```

```
enum e_dagType
{
        // two sequential statements                                DT_SEQ=0,

        // two sequential statements, and the value of the right
        // subtree is caried on.                                    DT_RSEQ,

        // two sequential statements which can be emitted in
        // abitrary order (for parameters)...                       DT_PSEQ,

        // used during building
        // should not appear in final dag                           DT_PLACEHOLDER,

        // a function                                               DT_FUNC,

        // a label                                                  DT_LABEL,

        // an unconditional jump                                    DT_JUMP,

        // jump if expression true                                  DT_JTRUE,

        // name of a global                                         DT_GLOBAL,

        // name of a formal                                         DT_FORMAL,

        // name of a local                                          DT_LOCAL,

        // dereference by a star                                    DT_DEREF,

        // perform function call                                    DT_CALL,

        // perform addition                                         DT_ADD,
        // perform subtraction                                DT_SUB,

        // perform multiplication                             DT_MUL,

        // perform arithmetic right shift                     DT_RSH,

        // perform arithmetic left shift                      DT_LSH,

        // perform bitwise and operation                      DT_BAND,

        // perform bitwise or operation                       DT_BOR,

        // perform bitwise nand operation                     DT_BNAND,

        // perform assignment                                 DT_GETS,

        // load integer constant                              DT_CONSTANT,

        // pass a parameter                                   DT_PASS,

        // return a value                                     DT_RETURN,

        // equality operator                                  DT_EQ,

        // inequality operator                                DT_NE,

        // greater than equal operator                        DT_GTE,

        // greater than operator                              DT_GT,

        // less than or equal operator                        DT_LTE,

        // less than operator                                 DT_LT,

        // short-circuit disjunction                          DT_AND,     DT_OR
};
typedef enum e_dagType          DagType;
```

```
struct s_dag
{
        int                     id;
        DagType         type;
        int                     refCount;
        DagAnnotation   notes;
        int             offset;
        Symbol          *mode, *symbol;
        struct s_dag            * left, *    right;
};
typedef struct s_dag    Dag;
```

B3. List of Some Important Functions

B4.1. Stack Functions

```
Stack * empty_stack();
Stack * push_stack(Stack *l, Dual v);
Stack * pop_stack(Stack *l, Dual *v);
Dual  top_stack(Stack *l);
void  free_stack(Stack *l);
int        inclusion_stack(Stack *l, Dual v, BstCallBack cmp);
```

B4.2. Binary Search Tree or Set Functions

```
Bst * insert_set(Bst *set, Dual val, BstCallBack cmp);
// Set-wise insertion into a bst
int        size_bst(Bst *rt);
Dual  first_bst(Bst *rt);
void  free_bst(Bst *bst);
Bst * copy_bst(Bst *orig);
Bst * merge_set(Bst *a, Bst *b, BstCallBack cmp);
Bst * remove_bst(Bst *bst, Dual v, BstCallBack cmp);
int        inclusion_bst(Bst *bst, Dual d); // uses default callback
// iterate over each element in the bst and stops if any of them
return non-zero.
// Return said non-zero value.
// user is passed as second parameter to call back function
Dual  each_bst(Bst *bst, Dual user, BstCallBack cb);
// perform binary search
// cb() should return as would comparison to guide the search.
// User will be passed as the second parameter to the callback fcn
Dual  search_bst(Bst *bst, Dual user, BstCallBack cb);
```

B4.3. Triangular Bit Matrix Functions

```
Tbm ** new_tbm(int width);
void free_tbm(Tbm **tbm, int width);
void add_tbm(Tbm **tbm, int x, int y);
int check_tbm(Tbm **tbm, int x, int y);
void remove_tbm(Tbm **tbm, int x, int y);
```

B4.4. Register Allocation Functions

```
void graph_color(Arena *a);
int scavenger(Arena *a); // to alert unused registers
```

B4.5. Optimizations

```
int perform_cse(Arena *a);
int perform_mem_optimization(Arena *a);
int perform_peephole(Arena *a);
int perform_dce(Arena *a);
int perform_loopunroll(Arena *a);
```

B4.6. Function Used During Code Generation

```
void quads2asm(Arena *a);
void sym2asm(Arena *a);
```

B4.7. Functions to Create and Iterate on Basic-blocks

```
Arena *new_arena();
void free_arena(Arena *a);
void dump_quads(Arena *a);
void dump_quad(Quad *cursor);
void remove_quad(Arena *a, Quad *q);
void insert_before(Arena *a, Quad *q, QuadType type, int dest, int sa,
int sb, Symbol *sym, int imm, int weight);
void insert_after(Arena *a, Quad *q, QuadType type, int dest, int sa,
int sb, Symbol *sym, int imm, int weight);
void insert_label(Arena *a, Symbol *label);
void insert_block(Arena *a);
void insert_line_number(Arena *a, int lin);
void insert_add(Arena *a, int dest, int sa, int sb, int weight);
void insert_sub(Arena *a, int dest, int sa, int sb, int weight);
void insert_and(Arena *a, int dest, int sa, int sb, int weight);
void insert_or(Arena *a, int dest, int sa, int sb, int weight);
void insert_nand(Arena *a, int dest, int sa, int sb, int weight);
void insert_sla(Arena *a, int dest, int sa, int imm, int weight);
void insert_sra(Arena *a, int dest, int sa, int imm, int weight);
void insert_ld(Arena *a, int dest, int sa, int weight);
void insert_st(Arena *a, int sa, int sb, int weight);
void insert_const(Arena *a, int dest, int imm, int weight);
void insert_offset(Arena *a, int dest, Symbol *label, int imm, int
weight);
void insert_jeq(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jne(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jgt(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jgte(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jlt(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jlte(Arena *a, int sa, int sb, Symbol *label, int weight);
void insert_jump(Arena *a, Symbol *label, int weight);
void insert_move(Arena *a, int dest, int sa, int weight);
void insert_mul(Arena *a, int dest, int sa, int sb, int weight);
void insert_jreg(Arena *a, int sa, int weight);
void insert_touch(Arena *a, int sourcereg);
void insert_same(Arena *a, int sa, int sb, Symbol *sm);
void recalculate_usage_counts(Arena *a);
void printQuad(Quad * temp);
void myPrintQuads(Arena * a);
char * quadToString(QuadType q);
```

```
void printQuadSimple(Quad * temp);

//This will look over an arena and assert that every temporary is
defined at most once. A debugging tool.
void assert_ssa(Arena *a);

// determine if the runtime values of temporaries $x and $y are
equivalent,
int cmp_runtime_vals(int x, Quad *rx, int y, Quad *ry);
void fix_jump_too_far(Arena *a);
```

B4.8. Function Used to Create and Access Symbol Table

```
void init_symtab(int count);
void finish_symtab();
void dump_symtab();
void dump_histograms();
void dump_offsets();
void dump_call_graph();
int check_recursion();
void note_call(Symbol *from, Symbol *to, Symbol *ret);
Symbol *lookup_symbol(const char *name);
Symbol *search_symbol(const char *name, Symbol *parent, SymType type);
Symbol *install_symbol(const char *name, SymType type);
Symbol *lookup_constant(int v);
// generate a new mode type as an n-ary array of something
Symbol * array_of(Symbol *mode, int n);
const char *mode2str(Symbol *mode);
// determine if two modes are compatible
int compatible_modes(Symbol *a, Symbol *b);

Symbol *create_special_global(const char *name, int p);
// get a structure by name
Symbol * mode_struct(char *name);
// get a union by name
Symbol * mode_union(char *name);
// get the integer mode
Symbol * mode_int();
Symbol * find_last_global();
// deprecated
int find_most_common_constant();
// assign constants to registers
void assign_constants_to_registers();
int get_element_size(Symbol *md);
int get_sizeof(Symbol *md);
// a lot of constants are used for accessing variables on the stack
frame.  Therefore, in order to keep our usage count of constants up-
to-date, we add n uses to each offset k, where n is the number of uses
of a variable at frame offset k.
void update_const_freqs_with_frame_offsets();
```

B4. Flow Control of The Code, Starting from the file Driver.c

Driver.c :

i. Contains the function main(). The series of operations done in main() are :
ii. Initialise the symbol table.
iii. Parse the file. If parsing error, then that is reported.
iv. Make sure main() is defined.
v. If all the components : Symbol Table, DAG etc. have been successful printed, dump them.
vi. Convert DAG to Quad by calling the function dag2quad(ast)
vii. DO graph coloring by calling graph_color(quads).
viii. If every variable, can be safely allotted a register, then just break off successfully, else
ix. Free some symbols in the registers and try again, increment the number of passes.
x. The first function called is dag2quad(asm) which returns a pointer to an Arena

B5. Modified Fields in ADT's and Functions in Jackcc Compiler to Accommodate Proposed Changes in the Thesis

B6.1. Fine Grain Thread Creation Pass

The structure Arena and Quad are appended with few more fields to create disjoint sub-block in fine grain thread extraction phase.

```
struct Arena
{
    Quad * first,* last;
    int *SubBlockIndex
    int *SubBlockSize;
    int num_quads
    int no_subblock;
};
```

```
struct Quad
{
   int dest
   int srcb, srca;
   int oper;
   int loop_weight;
   struct Quad *next,*prev;
};
```

In Arena, the instructions in corresponding basic block are stored as a linked list of Quads represented by first and last of the structure. Number of quads is stored in num_quads.
In Quad, the instruction stored is of the form dest = srca oper srcb where dest, srca, srcb are indices in the symbol table corresponding to the variables.

We divide the Arena into sub-blocks using disjoint-set operations based on true-dependencies. Instructions having true-dependency are grouped together. Let us assume that an arena has N instructions. To begin with, this arena is divided into N distinct sets (sub-blocks), each having one instruction. Two sets are combined into a collection of instructions (to form sub-blocks) if they have true-dependency. Structure of each sub-block is shown below.

```
struct sub_block
{
        Quad * q;
        struct sub_block * next;
        struct sub_block *sub_block_rep;
        int block_no;
        int task_no;
        int sub_block_no;
        int no_regs_block;
};
```

In the structure sub-block, we maintain the information task_no, sub_block_no, and block_no, which are task number, sub-block number of the block, and basic block number of that task of newly created sub-block. Above information are required by the instruction scheduler to schedule these sub-blocks on to different cores in out-of-order and produce the output in in-order. The no_reg_block will store the information of number of register required to the variables in that sub-block used by instruction scheduler and register allocation algorithm. The set_rep is the representative of a sub_block, used for disjoint union operation. We store the list of sub-blocks in an arena in the form of SetList S. Prototype of functions used to create disjoint sub-blocks.

```
//Approach 1 to Create sub-block

void makeSet(Arena A,List S);
void groupQuads(List S, number of quads);
void Union(List S,index i,index j);

//Approach 2 to Create sub-block

void Findind_Dominators(Basic Block Bp, sub-block List SList);
void Dominators_List();
void Dominniance_Frontier_calculation();
void Forcers_on_A_Node();
void PHI_function();
void Variable_renaming_Subblock_creation();
```

B6.2. Intra Block Scheduling

The function used for intra block scheduling is given below.

```
void mergeSubblock(List Sp,int  no_of_sub-blocks,int num_of_quads);
void scheduleBlock(Basic Block Bp, Hyper sub-block List HSList);
```

B6.3. Inter Block Scheduling

Next, we shall see the data structures and functions used to compute the Sub-block Dependency Graph (SDG). The data structure of a node of CFG is very similar to data structure of Arena (of previous section) with extra pointers to its children and parents in CFG. Data Structure of SDG is also given.

```
struct CFG
{
  Quad * first,* last;
  int SubBlockIndex[], SubBlockSize[];
  CFG *dependency_Block[];
  int dependency_SubBlock[][];
  int num_quads, no_subblock;
  int depcount;
  CFG *child1, *child2, *parents[];
  int no_parents;
  int *phi[];
  int subbl_phi[];
  int no_rows;
  SDG subBlockGraph[];
};
```

```
struct SDG
{
  Quad * first,* last;
  int SubBlockIndex
  int SubBlockSize;
  CFG *Block;
  SDG children[];
  SDG parents[];
  int no_parents;
  int no_child;
  int *phi[];
  int no_rows;
};
```

In structure for CFG, apart from the data elements present in Arena, we also have pointers child1, child2 and parents[], list of children and parents the CFG node can have. phi[] is the list of ϕ functions, each row corresponds to the function of one variable. It has as many columns as number of parents to the node. Each entry stores the version number of the variable from the corresponding branch. no_rows indicates the number of ϕ functions. Once we split the block into sub-blocks using Algorithm 1 of section 3, the array subbl_phi[] stores the sub-block number to which each ϕ function belongs. When we compute the dependencies of each sub-block, we store them in structures dependency_Block and dependency_SubBlock – while former stores pointers to blocks to which a sub-block is dependent upon, latter stores the corresponding sub-block number. depcount stores the number of dependencies. subBlockGraph[] is an array of nodes of sub-block dependency graph – one node corresponding to each sub-block.
The structure for SDG – a node in Sub-block Dependence Graph has list of quads, Sub-block number, size, list of children, list of parents, ϕ functions belonging to the sub-block and a pointer to the node of CFG to which the sub-block belongs.

```
//Approach 1 to SDG

ComputeDependency(Basic Block Bp, sub-block List SList);
SDG createSDG(Sub-block SBp, Sub-block SBq);

//Approach 2 to SDG
Variable_renaming_SDG_creation();

// Global Schedulers
InterBlockHIB(CFG C,SDG G);
InterBlockDBS(CFG C, SDG G);
InterBlockLLSF(CFG C, SDG G);
InterBlockMDS(CFG C, SDG G);
```

B6.4. Functions Used in Register Allocation

```
// Integrated Register Allocation and Scheduling

IntegratedScheulder(CFG C, SDG G);

// Register Allocation

MergeOperation(sub-block SBi , sub-block SBj);
void graph_color(Sub-block SBp);
```

List of References

[1] G. E. Moore, “*Cramming More Components Onto Integrated Circuits*", Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 -117, IEEE Solid-State Circuits Newsletter, vol. 11, no. 5, pp. 33 –35, September. 2006.

[2] R. Dennard “*Design of Ion-implanted MOSFETs With Very Small Physical Dimensions*” IEEE Journal of Solid State Circuits , vol. SC-9, no. 5, pp. 256-268, October. 1974.

[3] Gustafson, J." *Reevaluating Amdahl's Law*". Communications of the ACM 31(5), pp.532-533, 1988.

[4] H. Esmaeilzadeh, ‘‘*Dark Silicon and the End of Multicore Scaling*’’ 38th International Symposium. Computer Architecture, ACM Press, 2011.

[5] F. J. Pollack, “*New Micro-architecture Challenges In The Coming Generations of CMOS Process Technologies*” in Proceedings of the 32nd annual ACM/IEEE international symposium on Micro-architecture (MICRO 32), 1999.

[6] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. "*Clock rate Versus IPC: The End Of The Road For Conventional Microarchitectures*". In Proceedings of the 27th International Symposium on Computer Architecture , pp 248–259, June 2000.

[7] D. Geer, "*Chip Makers Turn To Multicore Processors*" IEEE Computer, vol. 38, no. 5, pp. 11-13, 2005.

[8] J. Held,, J. Bautista, and S. Koehl, “From a Few Cores to Many: A Tera-Scale Computing Research Overview,” white paper, Intel; http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf

[9] Andras Vajda, "*Programming Many-Core Chips*", Springer, pp 9-44, 2011.

[10] Sergio Saponara and Luca Fanucci, “Homogeneous and Heterogeneous MPSoC Architectures with Network-On-Chip Connectivity for Low-Power and Real-Time Multimedia Signal Processing,” VLSI Design, vol. 2012.

[11] M.D. Hill and M.R. Marty. "*Amdahl’s Law In The Multicore Era". IEEE Computer*, pp. 33–38, 2008.

[12] Dong Hyuk Woo, Hsien-hsin S. Lee, "*Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era"*, IEEE Computer, pp. 24-31, 2008.

[13] Quad-Core and Dual-Core Intel ® Xeon ® Processor 5000 Sequence and Intel ® 5100 Memory Controller Hub Chipset Development Kit User Guide.

http://download.intel.com/design/intarch/manuals/319157.pdf

[14] B.M.Rogers, A.Krishna, G.B.Bell, K.Vu, X.Jiang, and Y. Solihin, “Scaling The Bandwidth Wall: Challenges In and Avenues For CMP scaling,” SIGARCH Computer Architecture News , vol. 37, no. 3, pp. 371–382, 2009.

[15] Sergey Blagodurov , Sergey Zhuravlev , Alexandra Fedorova , Ali Kamali, " *A Case For NUMA-Aware Contention Management On Multicore Systems*", 19th international conference on Parallel architectures and compilation techniques, pp.11-15, September, 2010.

[16] Sun, X.H., Chen, Y. "*Reevaluating Amdahl's Law in the Multicore Era*". J. Parallel and Distributed Computing 70(2), pp.183-188, 2010.

[17] Sun, X.H., Chen, Y., Byna, S."*Scalable Computing in Multicore Era*". International Symposium on Parallel Algorithms, Architectures and Programming, 2008.

[18] Pai, V.S, Ranganathan, P, Adve, S.V, "*The Impact Of Instruction-Level Parallelism On Multiprocessor Performance and Simulation Methodology",* High-Performance Computer Architecture, 1997., Third International Symposium on , vol., no., pp.72-83, 1997.

[19] StallingsWilliam, "*Computer Organization and Architecture*, *Pearson Education*", 8th Ed., 2010.

[20] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve, "*The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors",* IEEE Transactions on Computers, vol. 48, pp. 218–226, February 1999.

[21] J.M.Tendler, J.S. Dodson, J.J.S.Fields, H. Le, and B. Sinharoy. "*Power 4 System Micro-architecture"*. IBM Journal of Research and Development, 46(1), pp 5-6, January 2002.

[22] P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, “*A Study Of The On-Chip Interconnection Network For The IBM Cyclops64 MultiCore Architecture*” International Parallel Distributed Processing Symposium, 2006.

[23] M. Taylor. The Raw Prototype Design Document. 2002. http://groups.csail.mit.edu/cag/raw/documents/RawSpec99.pdf.

[24] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim,M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. "*Baring It All to Software: Raw Machines"*. Computers, 30(9), pp 86-93, 1997.

[25] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S.W. Keckler, and D.C. Burger, "*The Distributed Microarchitecture of the TRIPS Prototype Processor*" *39th International Symposium on Microarchitecture (MICRO), December, 2006.*

[26] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D.C. Burger, and K.S. McKinley. "*Compiling for EDGE Architectures*" *International Conference on Code Generation and Optimization* (CGO), March, 2006.

[27] S. Muchnick. "*Advanced Compile Design and Implementation*". Morgan Kaufmann, 1997.

[28] R,Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. "*Efficient Computing Static Single Assignment Form and The Control Dependence Graph".* ACM Transaction on Programming Languages and Systems, 13(4),pp.451-490,1991.

[29] Y. Sazeides, S. Vassiliadis, J. E. Smith, *"The Performance Potential of Data Dependence Speculation& Collapsing"*, ACM/IEEE 29th International Symposium on Microarchitecture, pp.238-247,December 1996.

[30] D. Friendly, S. Patel, Y. Patt, "*Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors"*, ACM/IEEE 31st International Symposium on Micro-architecture, pp.173-181,November 1998.

[31] B. Ramakrishna Rau and Joseph A. Fisher. "*Instruction Level Parallel Processing: History, Overview, and Perspective*", Journal of *Supercomputing* 7, pp.9-50, May 1993.

[32] Dennis, J.B, Gao, G.R, "*An Efficient Pipelined Dataflow Processor Architecture"*, Supercomputing '88. [Vol.1]. Proceedings. , vol., no., pp.368-373, 14-18 November 1988.

[33] David A. Patterson, "*Reduced Instruction Set Computers"*, Communications of the ACM, v.28 n.1, p.8-21, January. 1985.

[34] Fisher, J.A. "*The VLIW Machine: A Multiprocessor for Compiling Scientific Code, Computer"* , vol.17, no.7, pp.45-53, July 1984.

[35] Smith, J.E, Sohi, G.S, "*The Micro architecture of Superscalar Processors"*, Proceedings of the IEEE , vol.83, no.12, pp.1609-1624, December 1995.

[36] R. Colwell, "*Architecture and implementation of a VLIW supercomputer*", International Conference on Super computing , pp 910–919, June 1990.

[37] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke, "*Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications*", International Symposium on High Performance Computer Architecture (HPCA), pp. 25-36, February 2007.

[38] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker, "A Distributed Control Path Architecture For VLIW Processors", 14th International Conference on Parallel Architectures and Compilation Techniques, pp. 197–206, September. 2005.

[39] Michael Chu and Scott Mahlke, "*Compiler-Directed Data Partitioning for Multicluster Processors*" 4th International Symposium on Code Generation and Optimization (CGO), pp. 208-218,Mar. 2006.

[40] Faraboschi P, Fisher J.A, and Young C, "*Instruction Scheduling For Instruction Level Parallel Processors"*, Proceedings of the IEEE , vol.89, no.11, pp.1638-1659, November 2001.

[41] Thomas L. Adam, K.M. Chandy and J.R. Dickson. "*A Comparison of List Schedules for Parallel Processing Systems"*. In Communications of the ACM volume 17 Issue 12, December 1974.

[42] Golumbic, M.C., and Rainish, V. "*Instruction Scheduling Beyond Basic Blocks"*. IBM Journal of Research and Development, Vol. 34, No. 1, January 1990.

[43] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth and Paul K. Rodman. "*A VLIW Architecture for a Trace Scheduling Computer"*. In ASPLOS-II Proceedings of the second international conference on Architectural support for programming languages and operating systems IEEE Computer Society Press Los Alamitos, CA, USA 1987.

[44] Lee, M, Tirumalai, P, Ngai, T.F, "*Software Pipelining and Superblock Scheduling: Compilation Techniques for VLIW Machines",* System Sciences, Proceeding of the Twenty-Sixth Hawaii International Conference on vol.1, 5, pp. 202- 213, 1993.

[45] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. "*Effective Compiler Support for Predicated Execution Using the Hyperblock"*. IEEE 25th International Symposium on Micro-architecture, pp. 45-54, December 1992.

[46] Cathy McCann , Raj Vaswani , John Zahorjan, "*A Dynamic Processor Allocation Policy for Multi-programmed Shared-Memory Multiprocessors*", ACM Transactions on Computer Systems (TOCS), v.11 n.2, pp.146-178,1993.

[47] A. M. Malik, J. McInnes, and P. van Beek. "*Optimal Basic Block Instruction Scheduling for Multiple-issue Processors Using Constraint Programming*". Technical Report CS-2005- 19, School of Computer Science, University of Waterloo, 2005.

[48] R. Gupta and M. L. Soffa, "*Region Scheduling: An Approach For Detecting and Redistributing Parallelism*" IEEE Transaction . Sofhyare Eng., vol. 16, April. 1990.

[49] Phillip B. Gibbons and Steven S. Muchnick, "*Efficient Instruction Scheduling for a Pipelined Architecture".* Proceedings of ACM SIGPLAN'86 Symposium on Compiler Construction. SIGPLAN Notices, 21 (7): 11-16, July 1986.

[50] David Bernstein, Micheal Rodeh, "*Global Instruction Scheduling of Superscalar Machine",* Conference on Programming Language Design and Implementation, Proceedings of ACM SIGPLAN, 241-255, 1991.

[51] David Bernstein and Izidor Gertner. "*Scheduling Expressions on a Pipelined Processor With a Maximal Delay of One Cycle*". *ACM Transaction. Programming Languages and System* 1, January, 1989.

[52] Jong-Jiann Shieh and Christos A. Papachristou. "*An Instruction Reorder For Pipelined Computers*". *In Proceedings of the 23rd annual workshop and symposium on Microprogramming and micro-architecture (MICRO 23)*. IEEE Computer Society Press, Los Alamitos, CA, USA, PP,135-142, 1990.

[53] M. Heffernan, K. Wilken and G. Shobaki. "*Data Dependency Graph Transformations For Superblock Scheduling*". In Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture, pp. 77-88, 2006.

[54] Thornton, J. E. "*Parallel Operation in the Control Data 6600*", Proceeding Fall Joint Computer. Conference, Part 2, Vol. 26, pp. 33 - 40, 1964.

[55] J. Hennessy and T. Gross, "*Postpass Code Optimization of Pipeline Constraints*" ACM Transaction. Programming Languages and Systems, vol. 5, no. 3, pp. 422-448, 1983.

[56] R. M. Tomasulo. "*An Efficient Algorithm for Exploiting Multiple Arithmetic Units*". IBM Journal of Research and Development. pp 25-33, 1967.

[57] C. Chekuri, R. Motwani, R. Johnson, B. Ramakrishna Rau, B. Natarajan. "*Profile-Driven Instruction Level Parallel Scheduling",* HP Laboratories Technical Report, HPL-96-16, January 1996.

[58] A. Fisher, "*Trace Scheduling: A Technique for Global Microcode Compaction*" IEEE Transaction. Computers, vol. 30, no. 7, pp. 478-490, July 1981.

[59] M. Lam. "Software Pipelining: An Effective Scheduling Technique For VLIW Machines". *SIGPLAN Not,* June 1988.

[60] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. "Enhanced Modulo Scheduling For Loops With Conditional Branches". *25th annual international symposium on Microarchitecture* (MICRO 25). IEEE Computer Society Press, pp.170-179, 1992.

[61] S. A. Mahlke. "*Sentinel Scheduling: A Model For Compiler-Controlled Speculative Execution*" ACM Transaction Computer and System., vol. 11, pp. 376–408, November. 1993.

[62] F. W. Burton, "*Speculative Computation, Parallelism, and Functional Programming*" IEEE Transaction. Computation. vol. C-34, pp.1190–1193, 1985.

[63] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "*The Superblock: An Effective Technique for VLIW and Superscalar Compilation*" J. Super-computing, vol. 7, no. 1, pp. 229-248, 1993.

[64] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. "*Generalized Instruction Selection Using SSA-graphs*". *ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems* (LCTES '08). pp.31-40, 2008.

[65] McConnell, C. and R. Johnson. "*Using Static Single Assignment Form in a Code Optimizer"*. ACM Letters on Programming Languages and Systems, 1(2), p. 152-160, 1992.

[66] J. Donald and M. Martonosi, "*Techniques for Multicore Thermal Management: Classification and New Exploration", SIGARCH Computer Architecture News,* vol. 34, no. 2, pp. 78-88, 2006.

[67] Bernhard Scholz and Erik Eckstein. "*Register Allocation for Irregular Architectures*". In LCTES/SCOPES, ACM, pp 139–148, 2002.

[68] Timothy Kong and Kent D Wilken. "*Precise Register Allocation for Irregular Architectures*". In International Symposium on Micro-architecture, ACM, pp 297–307, 1998.

[69] David Koes and Seth Copen Goldstein. "*A Progressive Register Allocator for Irregular Architectures*". In CGO, pp 269–280, 2005.

[70] G. Chaitin. "*Register Allocation and Spilling via Graph Coloring*". In Proceedings of the SIGPLAN Symposium on Compiler Construction, pp 98-105, June 1982.

[71] Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. "*Coloring Heuristics for Register Allocation*". In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 275-284, 1989.

[72] Fredrick Chow and John Hennessy, "*Register Allocation by Priority-based Coloring*", Proceedings of the ACM SIGPLAN Symposium on Compiler Construction SIGPLAN Notices Vol. 19, No. 6, June 1984.

[73] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai, "*Fusion-Based Register Allocation*", ACM Transactions on Programming Languages and Systems, Vol. 22, No. 3, pp 431-470, May 2000.

[74] Massimiliano Poletto and Vivek Sarkar, "*Global Linear Scan Register Allocation*", ACM Transactions on Programming Languages and Systems, Vol. 21, No. 5, pp 895-913, September 1999.

[75] Hanspeter Mossenbock and Michael Pfeiffer. "*Linear Scan Register Allocation in the Context of SSA Form and Register Constraints*". In CC, LNCS, pp 229–246, 2002.

[76] Cindy Norris and Lori. L. Pollock, "*Register Allocation over the Program Dependence Graph*". SIGPLAN 94-6/94.

[77] Rajiv Gupta, Mary Lou Soffa, Denise Ombres, "*Efficient Register Allocation via Coloring Using Clique Separators*", ACM Transactions on Programming Languages and Systems, Vol 16, No 3, pp 370- 386, May 1994.

[78] David Callahan, Brian Koblenz, "*Register Allocation via Hierarchical Graph Coloring*". In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation Toronto Ontario, Canada, pp. 26-28, June, 1991.

[79] Changqing Fu, Kent Wilk, "*A Faster Optimal Register Allocator*", Proceedings of the 35th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-35), 2002.

[80] Rainer E. Burkard, Eranda C¸ ela, Panos M. Pardalos, and Leonidas S. Pitsoulis. "*Quadratic Assignment Problems*". European Journal of Operational Research, 15:pp 283–289, 1984.

[81] Todd A. Proebsting and Charles N. Fischer, "*Probabilistic Register Allocation*", ACM SIGPLAN ’92 PLD1-6/92/CA.

[82] Christian Wimmer and Michael Franz, "*Linear Scan Register Allocation on SSA Form*". CGO, pp. 24–28, April, 2010.

[83] S. Hack and G. Goos. "*Optimal Register Allocation for SSA-form Programs in Polynomial Time*". Information Processing Letters, 98(4):pp.150–155, 2006.

[84] Fernando Magno Quintao Pereira and Jens Palsberg. "*Register Allocation After Classic SSA Elimination Is NP-Complete*". In Foundations of Software Science and Computation Structures. Springer, 2006.

[85] Fernando Magno Quintao Pereira and Jens Palsberg. "*Register Allocation via Coloring of Chordal Graphs*". In APLAS, Springer, pp 315–329. 2005.

[86] David A. Berson, Rajiv Gupta, and Mary Lou Soffa, "*Integrated Instruction Scheduling and Register Allocation Techniques*", LCPC’98, LNCS 1656, pp. 247–262, 1999.

[87] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. "*Combining Register Allocation and Instruction Scheduling*", Stanford Reports, 1995.

[88] Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. "*Integrated Pre Pass Scheduling For a Java Just-In-Time Compiler On The IA-64 Architecture*". In Proceedings of the international symposium on Code generation and optimization, pp. 159-168, 2003.

[89] Cindy Norris and Lori L. Pollock. "*A scheduler Sensitive Global Register Allocator*". *Proceedings of Supercomputing’93*, pages 804-813, pp.248-253, 1993.

[90] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. "*URSA: A Unified Resource Allocator for Registers and Functional Units in VLIW Architectures*". IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, pages 243–254, 1993.

[91] Walter Lee , Rajeev Barua , Matthew Frank , Devabhaktuni Srikrishna , Jonathan Babb , Vivek Sarkar , Saman Amarasinghe. "*Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine"*. In Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems. pp. 46-57, 1998.

[92] H. Zhong, S. A. Lieberman, and S. A. Mahke, "*Extending Multicore Architectures To Exploit Hybrid Parallelism In Single-Thread Applications*," International Symposium. High Performance Computer Architecture , pp. 25–36, March, 2007.

[93] Michael Chu, Rajiv Ravindran, and Scott Mahlke. "*Data Access Partitioning For Fine-grain Parallelism on Multicore Architectures*". In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture* (MICRO 40). pp 369-380, 2007.

[94] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa, "*Fine-Grain Parallelism Using Multi-Core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function*" International Conference on Parallel Processing, pp. 9-17, 2009.

[95] Muthu Manikandan Baskaran , Nagavijayalakshmi Vydyanathan , Uday Kumar Reddy Bondhugula , J. Ramanujam , Atanas Rountev , P. Sadayappan, "*Compiler Assisted Dynamic Scheduling for Effective Parallelization Of Loop Nests On Multicore Processors*", Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, February 14-18, 2009.

[96] Yong Li , Ahmed Abousamra , Rami Melhem , Alex K. Jones, "*Compiler Assisted Data Distribution for Chip Multiprocessors*", Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pp 11-15, September, 2010.

[97] George C. Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. "*Resource-Aware Compiler Prefetching for Many-Cores*". *Ninth International Symposium on Parallel and Distributed Computing* (ISPDC '10). IEEE, pp 133-140, 2010.

[98] Vinay G. Vaidya, Priti Ranadive and Sudhakar Sah, "*Dynamic Scheduler for Multicore Systems*" 2nd International Conference on Software Technology and Engineering, Puerto Rico, USA, pp. 13-16, 2010.

[99] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. "*Contention-Aware Scheduling on Multicore Systems*". *ACM Transaction Computing System.* December 2010.

[100] Mojtaba Mehrara. *"* Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications*"*. PhD. Dissertation. University of Michigan, Ann Arbor, MI, USA. (ACM) 2011. http://cccp.eecs.umich.edu/theses/mehrara-thesis.pdf

[101] Yuanrui Zhang, Jun Liu, Emre Kultursay, Mahmut Kandemir, Nikos Pitsianis, and Xiaobai Sun. *"Automatic Parallel Code Generation for Data Translation on Multicore"*. Journal of Circuits, Systems and Computers, World Scientific, 2012.

[102] Cheng-Yen Lin; Chi-Bang Kuan; Jenq Kuen Lee, *"Compilers for Low Power with Design Patterns on Embedded Multicore Systems"* Journal of Signal Processing Systems, Springer, US, 1-17, 2014

[103] Shaoming Chen, Vue Ru, Ying Zhang, Lu Peng, Jesse Ardonne, Samuel Irving, Ashok Srivastava. "*Increasing Off-Chip Bandwidth in Multicore Processors with Switchable Pins*", 41st annual international symposium on Computer architecture (ISCA '14). IEEE Press, pp. 385-396, 2014.

[104] The Jack Compiler, http://jackcc.sourceforge.net.

[105] J. Babb , M. Frank , V. Lee , E. Waingold , R. Barua , M. Taylor , J. Kim , S. Devabhaktuni , A. Agarwal, "*The RAW Benchmark Suite: Computation Structures For General Purpose Computing"*, Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, pp.134, 1997

[106] The Raw Benchmark Suit, http://groups.csail.mit.edu/cag/raw/benchmark/

[107] Cormen, T., Leiserson, C., and Rivest, R. "*Introduction to Algorithms*". The MIT Press Cambridge MA 2001.

[108] C, Cascaval, J. Castanos. L. Ceze, M. Denneau, M. Gupta, D. Lieber, J.E. Moreira, K. Strauss, and H. S. W. Jr. "*Evaluation of Multithreaded Architecture for Cellular Computing"*. In proceedings of the 8th International Symposium on High Performance Computer Architecture, page 311-322, January 2002.

List of Publication by Author

Conferences

[01] D.C. Kiran, S. Gurunarayanan, and J.P.Misra, "*Taming Compiler to Work with Multicore Processors"*. IEEE Conference on Process Automation, Control and Computing. 2011.

[02] D.C.Kiran, B. Radheshyam. S. Gurunarayanan, and J.P.Misra, "*Compiler Assisted Dynamic Scheduling for Multicore Processors"*. IEEE Conference on Process Automation, Control and Computing. 2011.

[03] D.C. Kiran, S. Gurunarayanan, J.P.Misra and Faizan Khaliq, "*An Efficient Method to Compute Static Single Assignment Form for Multicore Architecture"*. In 1st IEEE International Conference on Recent Advances in Information Technology, pp. 776-789, March 2012.

[04] D.C. Kiran, S. Gurunarayanan, and J.P.Misra, "*Compiler Driven Inter Block Parallelism for Multicore Processors"*. In 6th International Conference on Information Processing, published in the Communications in Computer and Information Science (CCIS), Springer-Verlag, August 2012.

[05] D.C. Kiran, S. Gurunarayanan, Faizan Khaliq, and Abhijeet Nawal, "*Compiler Efficient and Power Aware Instruction Level Parallelism for Multicore Architectures"*. In The International Eco-friendly Computing and Communication Systems, published in the Communications in Computer and Information Science (CCIS), Springer-Verlag, pp.9-17 August 2012.

[06] D.C. Kiran, S. Gurunarayanan, J.P.Misra, and D.Yashas "*Integrated Scheduling and Register Allocation For Multicore Architecture*". In IEEE Conference on Parallel Computing Technologies PARCOMPTECH-2013, Organized by C-DAC in IISC Bangalore, February 2013.

[07] Munish Bhathia, D.C.Kiran, S Gurunarayanan, and J.P.Misra, "*Fine Grain Thread Scheduling on Multicore Processors: Cores With Multiple Functional Units*". ACM Compute. Aug 2013.

Journals

[01]. D. C. Kiran, S. Gurunarayanan, Janardan Prasad Misra, and Abhijeet Nawal, “Global Scheduling Heuristics for Multicore Architecture,” Scientific Programming, vol. 2015, Article ID 860891, 12 pages, 2015. doi:10.1155/2015/860891.
http://www.hindawi.com/journals/sp/2015/860891/cta/ (This paper is sited in ACM Digital library)

[02]. D.C. Kiran, S. Gurunarayanan, J.P.Misra & Munish Bhathia "Register Allocation for Fine Grained Threads on Multicore Processors". Journal of King Saud University - Computer and Information Sciences, Elsevier (Accepted) To appear in Volume 27, Issue 3 2015.

PhD. Forum

[01] D.C. Kiran "*Compiler Support and Optimization for Multicore Processors*" PhD Forum - organized in conjunction with ICDCN 2013 at Tata Institute of Fundamental Research, Mumbai, January 3-6, 2013.

Biographies

Brief Biography of the Candidate

Mr. D.C.Kiran is a faculty in department of Computer Science and Information Systems at Birla Institute of Technology and Science Pilani (BITS-Pilani), Pilani campus, India. He obtained his B.E (CSE) from VTU Karnataka in 2002, and M.E (CSE) from Anna University Chennai in 2004. He is teaching at BITS-Pilani since 2005. His research and teaching interests include Compiler Construction & Optimization, Programming Language Design and Computer Architecture. He has published his works in two national and six international conferences. Contact him at dck@pilani.bits-pilani.ac.in.
http://universe.bits-pilani.ac.in/pilani/dck/profile

Brief Biography of the Supervisor

Professor S Gurunarayanan is a Professor in the department of Electrical and Electronics Engineering and Dean of Work Integrated Learning Program Division at Birla Institute of Technology and Science Pilani (BITS-Pilani), Pilani campus, India.
He obtained his M.E and Ph.D from BITS-Pilani.
He is with BITS-Pilani since 1987.
His research and teaching interests include VLSI Design, Digital Design, Embedded Systems and Computer Architecture.
Contact him at sguru@pilani.bits-pilani.ac.in.
http://universe.bits-pilani.ac.in/Pilani/sguru/profile