Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 860891, 12 pages
http://dx.doi.org/10.1155/2015/860891

Research Article

Global Scheduling Heuristics for Multicore Architecture

D. C. Kiran,[1] S. Gurunarayanan,[2] Janardan Prasad Misra,[1] and Abhijeet Nawal[3]

[1]*Department of Computer Science and Information Systems, Birla Institute of Technology and Science Pilani, Rajasthan 333031, India*
[2]*Department of Electrical Electronics and Instrumentation, Birla Institute of Technology and Science Pilani, Rajasthan 333031, India*
[3]*Oracle India Pvt. Ltd., Bangalore, Karnataka 560076, India*

Correspondence should be addressed to D. C. Kiran; dckiran@gmail.com

Received 22 July 2014; Revised 26 March 2015; Accepted 27 April 2015

Academic Editor: Jan Weglarz

This work discusses various compiler level global scheduling techniques for multicore processors. The main contribution of the work is to delegate the job of exploiting fine grained parallelism to the compiler, thereby reducing the hardware overhead and the programming complexity. This goal is achieved by decomposing a sequential program into multiple subblocks and constructing subblock dependency graph (SDG). The proposed schedulers select subblocks from the SDG and schedules it on different cores, by ensuring the correct order of execution of subblocks. In conjunction with parallelization techniques, locality optimizations are performed to minimize communication overhead between the cores. The results observed are indicative of better and balanced speed-up per watt.

1. Introduction

Multicore or on-chip multiprocessor is the technique to build high performance and energy efficient microprocessors. In general, computing systems using a multicore processor do not directly result in faster execution and reduced power consumption. To fully exploit the architectural capability and inherent fine grained parallelism of an application, it is desired to have parallel program. Writing parallel program is tedious and requires expertise. It is essential to provide compiler level support for converting the preexisting sequential program to parallel implementations such that it can be scheduled on multiple cores. Parallelization can be achieved in the following ways:

(i) Allow programmers to use parallel programming constructs to explicitly specify which parts of the program can run in parallel.

(ii) Allow operating system (OS) to schedule different tasks on different cores.

(iii) Allow hardware to extract parallelism and schedule it dynamically.

(iv) Allow the compiler to extract parallelism and schedule it.

In the first approach, developing and verifying an explicitly parallel program are expensive and do not scale with the number of cores [1]. In the second approach, the operating system realizes each core as a separate processor and OS scheduler schedules coarse grain threads onto different cores. In the thread style approach, two explicit parallel primitives are independent unless explicit communication primitives (for synchronization) are added to stress what is inside the original program. Further the multicore processor architecture differs from traditional multiprocessors in terms of having shared caches, memory controllers, smaller cache sizes available for each computational unit, and low communication latency between cores [2]. Owing to the architectural difference, it is desirable to extract fine grained thread and schedule it onto multiple cores instead of scheduling coarse grained thread as done in multichip multiprocessing systems (SMP system).

In hardware-centric approach, the hardware dynamically detects opportunities for parallel execution and schedules operations to exploit available resources [3]. This approach adds more circuits, resulting in complex hardware implementations of algorithms such as branch prediction, instruction level parallelism detection, and register renaming. The hardware based approach works under heavy resources and time constraints.

In software-centric approach, a compiler analyzes the program for the possibilities of parallelism, identifies the segments of the program which could be executed in parallel, and uses suitable scheduler to schedule the parallel constructs onto multiple cores. Using the various kinds of dependency analysis, the compiler can identify the independent instructions that can run in parallel [4]. The compilation being offline, one time activity, rigorous analysis to achieve the optimal amount of program parallelization can be carried out.

The aim of our research is to provide compiler support to exploit parallelism by extracting fine grained threads from a sequential program and creating schedules for multiple cores. The following challenges in multicore environment are addressed.

(i) Parallelizing sequential program and scheduling.

(ii) Memory management and data communication:

  (a) Register allocation.
  (b) Memory bandwidth.
  (c) Locality of reference.
  (d) Memory contention.

(iii) Power consumption.

Herein, four-compiler level global scheduling heuristics has been proposed and speed-up performance for each of the approaches has been compared. For each of the techniques, the intercore communication cost has been considered and an effort has been made to reduce the communication cost. Further, the power consumption for each of the techniques is evaluated and compared.

The paper is organized as follows. Section 2 gives an overview of different parallel architectures and their relation with the compiler. Section 3 gives a detailed background of the proposed work. An example illustrating the steps involved in the proposed algorithm is presented in Section 3. Section 4 provides the detailed description of the proposed global scheduling algorithms. A brief description of the compiler and the model of the multicore architecture used in the work is given in Section 5. Analysis and discussion of the results are given in Sections 6, 7, and 8, respectively. Finally, Section 9 concludes, summarizing the main achievements of our work.

2. Relation between ILP, Parallel Processor Architectures, and Compiler

To achieve high performance computing, a single core processor with parallel processing features was 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 become an important factor. The success of every 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 factors in the design of high-performance computers. Parallelism comes in different forms, namely, instruction level parallelism (ILP), task/thread level parallelism (TLP), memory level parallelism, and so forth. A compiler was used by the parallel architectures to exploit parallelism as required by them to squeeze more performance.

This section 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.

Instruction level parallelism (ILP) is used for speeding up the execution of a program by allowing a sequence of instructions derived from a sequential program to be parallelized for execution [3]. The basis of ILP is dependency analysis. The result of this analysis is used to identify the independent instructions which can be executed in parallel. The instruction dependency is of three types, the name dependency, the control dependency, and the data dependency. There are two types of name dependencies, Write after Write dependency (WAW) or antidependency and Write after Read dependency (WAR) also known as output dependency which 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. A 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 dependency (RAW) the true dependency that falls under data dependency category can be removed at run time using dynamic optimizations like data collapsing [5] and reassociation [6]. These methods need extra hardware to implement the optimization which violates the basic philosophy of RISC architecture to reduce hardware and power consumption. Also, any optimization done at runtime has a direct impact on the performance of the processor.

In the past, remarkable advancement has been made in the design of parallel architecture, and the compiler is used for exploiting ILP on such architectures [7]. The nature of compiler support for the processor architecture is an indenture and varies across design philosophy.

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 superscalar processors are the three existing techniques. In pipelined processor, a task is broken into stages and is executed on different (shared) processing units by overlapping the execution of instructions in time [8, 9]. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation could be continued without interruption throughout program execution. Unfortunately, the processor sometimes

stalls as a result of data dependency and branch instructions. The RISC solution to this problem is instruction reordering. The task of instruction 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 [10]. The major drawbacks of VLIW are its high instruction storage requirement and high instruction memory bandwidth requirements, due to the horizontal nature of instruction and its fetch frequency. Further, programs compiled for one VLIW machine are not compatible with another generation of a processor family. Superscalar processors are those which allow several instructions to be initiated simultaneously and executed independently. These processors have ability to initiate multiple instructions during the same clock cycle [11]. For proper utilization of a superscalar machine of degree n, we must have n instructions that can execute in parallel at all times. If an instruction-level parallelism of n is not available, stalls and dead time will occur forcing the instructions to wait for the results of prior instructions. Another major drawback with the superscalar processor is that all the execution units share the memory and register file leading to more register spilling and race condition.

VLIW and superscalar machines both benefit from instruction reordering resulting in better ILP. In VLIW, all dependencies are checked during compile time, and the search of independent instructions and scheduling is done exclusively by the compiler. The hardware has no responsibility for 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.

In multicore architectures the compiler requires to go beyond exploiting ILP in the innermost loops as has usually been the case in traditional ILP compilers. Also, the multicore system has some spatial requirements. First, when a value is live across the scheduling region, its definition and uses must be mapped to the same core to reduce communication cost. Second, the instructions like load and store must access the memory banks like a personal cache for performance reason and to avoid race condition.

2.1. Instruction Scheduling. In superscalar and VLIW machines, the only scheduling issue is a parallelism which is bonded with temporal issues [12, 13]. ILP is achieved in presence of register pressure. The spatial problem is neglected as these architectures exchange the shared/dependent operands through a shared register file which is absent in multicore systems. In pipeline based machines, scheduler reorders instructions in such a way that minimizes pipeline stalls. The rescheduling 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 include list scheduling, trace scheduling, superblock scheduling, and hyperblock scheduling. All these scheduling techniques are classified based on the nature of the control flow graph they use, that is, 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 [14] and is based on highest level priority scheme. Trace scheduling and superblock and hyperblock scheduling are global scheduling techniques that work on regions known as traces which consist of a contiguous set of basic blocks [15]. Trace scheduling combines the most common trace of basic blocks and schedules them as a single block [16]. Superblock scheduling is the same as trace scheduling without side entrances [17]. Hyperblock scheduling combines basic blocks obtained from multiple paths of control flow graph [18].

In run-time scheduling, an instruction is issued after it is decoded and when its operands are available [19]. The run-time scheduling mechanisms exhibit adaptive behavior which leads to a higher degree of load balancing. The run-time scheduling policies incur high run time overhead, which may lead to degradation of 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 a 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.

All the existing scheduling techniques are unable to make optimal choices as scheduling algorithms are strongly dependent on the machine model. The instruction scheduling techniques are NP-complete and follow heuristics [20]. Some heuristics use loop transformations, static branch prediction, speculative code motion, predicated execution, software pipelining, and clustering [21]. Different heuristics work well with different types of graphs.

3. Background of the Proposed Work

This section introduces the background of the proposed global scheduling techniques. The proposed work involves

(i) Parallel region formation or extracting fine grained threads [22–24].

(ii) Local scheduler to schedule fine grained threads onto multiple cores [25, 26].

Two additional passes are introduced into the original flow of compiler as shown in Figure 1: fine grained thread extractor pass and scheduler pass. The fine grained thread extractor module acts upon the basic blocks (B_p) of control flow graph

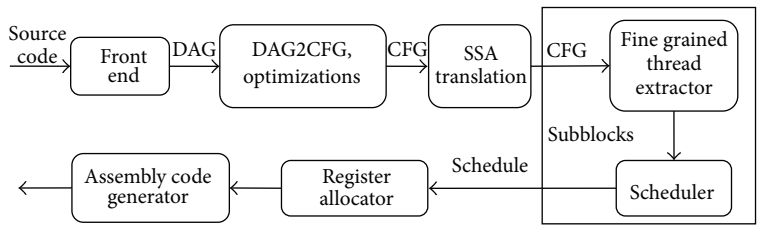
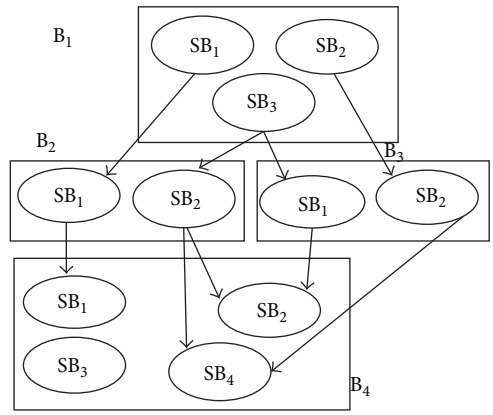


Figure 1: Modified flow of compiler.



(a)

	Subblocks	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 2: (a) Subblock dependency graph. (b) Dependency matrix.

(CFG). The instructions in each basic block of CFG are analyzed for dependency to create disjoint subblocks. The instructions in the subblocks are in SSA form and are true dependent. This ensures the spatial and temporal locality. In Figure 2(a), the CFG has four basic blocks B_1, B_2, B_3, and B_4. The disjoint set operations are applied to each basic block to form subblock SB_i. The subblock SB_i belonging to basic block B_p is referred to as SB_iB_p. To facilitate global scheduling new data structure called subblock dependency graph (SDG) is proposed. The SDG is a graph $G(V, E)$, where vertex $v_i \in V$ is a subblock 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 $p \neq q$. The total number of out-edges from a vertex (subblock) in SDG is referred to as degree of dependency. We say that SB_jB_q is, AFTER related to SB_iB_p. SDG is represented as dependency matrix in Figure 2(b). In dependency matrix all subblocks are arranged in the first column and the rest of the column entry indicates the dependency list. If the subblock SB_jB_q is dependent on subblock SB_iB_p, then SB_iB_p is added to the dependency list of SB_jB_q, meaning that SB_jB_q should be scheduled only after SB_iB_p completes its execution. The subblock SB_jB_q can be scheduled only if the list is empty; otherwise it should wait till the list becomes empty. Efforts are made to reduce the compilation time for performing fine grained extraction pass. The subblocks are created such that they ensure spatial and temporal locality.

The CFG shown is the output of the fine grained extractor pass and SDG formation phase of the compiler. The subblock SB_1 of basic block B_2 is AFTER related to SB_1 of block B_1 and it is denoted by $SB_1B_1 \rightarrow SB_1B_2$ and similarly $SB_2B_1 \rightarrow SB_2B_3$. The corresponding dependency matrix is shown in Figure 2(b).

The fine grained threads can be scheduled using scheduler. The scheduler ensures that the subblocks scheduled on different cores at the same time will not communicate nor access the same data, thus providing lock-free synchronization.

The scheduling pass follows the SSA translation and fine grained thread extraction pass in the proposed compiler framework. The disjoint subblocks created in the subblock creation phase are scheduled onto the cores. The local scheduling is coined as *intrablock scheduling* (IBS), where only subblocks inside a basic block are considered for scheduling. The results of IBS are compared with the results of global scheduling algorithm in the later section.

4. Global Scheduling Heuristics

The global scheduling heuristics identify the subblocks across the CFG that can execute in parallel on different cores. This is termed as *interblock scheduling*. The subblocks are disjoint within a basic block of CFG, but the subblocks across the basic blocks need not be disjoint. The nondisjoint subblocks should be executed sequentially. Interblock parallelism is a process of finding the nondisjoint subblocks across the basic blocks.

In this section, four novel global scheduling heuristics are proposed to achieve the conflicting goals such as high performance, low communication cost, high performance per power, and scalability. The novelty of the proposed algorithms is their efficient scheduling strategies which yield better solutions without incurring a high complexity.

The proposed global scheduling algorithms use the subblock dependency graph (SDG) to schedule subblock onto 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 subblock in the SDG. It is a linear-time algorithm which uses an effective search strategy to schedule the subblock onto the core with minimum schedule time. The second algorithm, called Dependent Subblock Based (DSB) scheduler, is based on the scheduling latency and subblocks dependency. All the paths from a given block to leaf node of SDG are identified and schedule latency for each path is computed. The subblocks in the path with the highest schedule latency are chosen for scheduling on different cores. The third algorithm is a Maximum Dependency Subblock (MDS) that calculates priority of the subblock based on maximum dependencies and minimum execution time. The fourth algorithm is the Longest Latency Subblock First (LLSF) which schedules considering only latency of the subblock. The proposed algorithms have been evaluated through extensive experimentations and yield consistently better performance when compared with numerous existing algorithms.

The aim of all the four proposed global scheduling heuristics is to create correct schedules that exploit the computing competence of multicore processor fully by decreasing the execution time of the program. It is desired to have compile time efficient scheduling heuristics along with being capable of creating correct schedule. The four heuristics differ in compilation time but do not compromise with speed-up. The data structure requirement of each heuristic is different and this requirement drives the compilation time. Based on the nature of the program, that is, degree of dependency between the subblocks of the program, the data structure requirement changes. Thus, based on the nature of the program being compiled, the best suitable scheduler may be used to create correct schedule efficiently. The HIB scheduler is suitable when the SDG is not too dense; that is, subblocks in SDG have fewer children and efficient techniques to traverse the tree that can be used. The DSB and MDS schedulers try to schedule dependent subblocks onto same core to reduce the communication cost. These schedulers are suitable when SDG is dense and benefits only when fewer cores are used by applying the power optimization technique. DSB is compile time efficient than MDS as the latter spends time in prioritizing the subblock before creating schedule. The LLSF scheduler has advantage over the other proposed global schedulers in terms of speed-up but has penalty of communication cost and power. This scheduler can be used in an environment where performance is crucial and no other optimization is required. A detailed discussion on selecting the best suitable scheduler for the program being compiled is given in Section 8.

In general, the global scheduler selects the subblock i of basic block B_p (SB_iB_p) from the subblock dependency matrix if its dependency list is empty. Once SB_iB_p is scheduled and completes its execution, its entry is removed in all dependency lists. The decision of scheduling a subblock on a core is based on the invariants such as scheduling latency, computed ready time (TRdy), and finish time (TFns) of the subblock SB_iB_p. The schedule time (TSch) of subblock and total scheduled time of core (TSct) are also taken into consideration to check the availability of a core to schedule the subblocks. Height and schedule latency of subblocks 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 subblocks currently scheduled on it. TSct is computed in top-down fashion on SDG. TSct suggests the time at which next subblock could be scheduled onto the core. The ready subblock is scheduled on a core with lower TSct.

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

$$\text{Height}_i = \text{Maximum}\left(\text{Height}\left(\text{AIS}\right)\right) + 1. \tag{1}$$

Equation (2) gives the predicted finish time (TFns) of a subblock SB_iB_p:

$$\text{TFns}_i = C_i + \text{TSch}_i, \tag{2}$$

where C_i is a total cycle time of ith subblock and TSch_i is schedule time of ith subblock.

The ready time (TRdy) of a subblock SB_iB_p is given below in (3). Ready time of a subblock is the time at which subblock is free from all its dependencies and ready to be scheduled on a core, that is, maximum finish time of all its immediate predecessors (AIP):

$$\text{TRdy}_i = \text{Maximum}\left(\text{TFns}\left(\text{AIP}\right)\right). \tag{3}$$

The schedule latency (L_i) of a subblock is given in (4). The schedule latency of leaf subblock in SDG is the total number of instructions in the leaf subblock. The schedule latency of SB_iB_p is a sum of maximum latency of all its immediate successors (AIS) and total number of instructions inside the subblock SB_iB_p or total cycle time of ith subblock SB_iB_p:

$$L_i = \text{Maximum}\left(L\left(\text{AIS}\right) + C_i\right). \tag{4}$$

The total scheduled time of a core k (TSct) is given in

$$\text{TSct}_k = \text{TSct}_{k-1} + C_i, \tag{5}$$

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

4.1. Height-Instruction Count Based (HIB). The HIB scheduler uses the subblock dependency graph represented in the form of the adjacency matrix (subblock dependency matrix) to take scheduling decisions.

Algorithm 1 (Height-Instruction Count Based scheduler).

(1) Calculate height of each subblock

Height of subblock i = Maximum ((height of all immediate successors) + 1).

(2) Initialize a priority queue

Q = {All head nodes, that is, nodes having only outgoing edges}.

(3) Schedule:

(a) If single core

(i) Remove highest priority node from queue.
(ii) Insert those nodes which are ready to schedule after scheduling the highest node in the ready queue.
(iii) Schedule the node on core.
(iv) Repeat the same process until queue gets empty.

(b) If multiple cores

(i) Repeat steps (ii) to (v) until queue gets empty.
(ii) Select a core with minimum TSct.
(iii) Select a node with the highest priority.
If TRdy of all nodes present in the queue is greater than the current TSct, then insert 1 free cycle.
Go to step (ii).
(iv) Schedule node on core and increment current core time.
(v) Update TFns of this node and TRdy of all its immediate successors.
(vi) Place its immediate successors in the queue if they are ready to schedule and revise the priorities of old nodes according to the priorities for new nodes.
Go to step (i).
(vii) END.

The scheduler creates a priority queue using SDG and schedules the subblocks onto multiple cores. The scheduler will schedule the subblock with the highest priority in the priority queue on the core with minimum Tsct. The scheduler updates the priority queue with new subblocks and removes their entry in subblock dependency matrix. A subblock can be added to priority queue if the dependency list of that subblock is NULL. The priority of the node is computed based on height and instruction count. The node at the highest level and having more instruction count is given the highest priority. The information required by HIB scheduler is computed using (1), (2), (3), and (5).

4.2. Dependent Subblock Based (DSB). The DSB scheduler collects all the subblocks and stores them in nonincreasing order of their latency. Initially the scheduler picks the subblock with the highest schedule latency and schedules it onto any one of the cores. Later scheduler picks the immediate ready successor of the previously scheduled subblock in the SDG. The successor subblock is scheduled onto 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 subblock, the TSct of the core is updated. The advantage of scheduling dependent subblocks onto the same core is the reduced communication between the core.

The information required by DSB scheduler is computed in (2), (3), (4), and (5).

Algorithm 2 (Dependent Subblock Based scheduler).

(1) Find latency.

(2) Sort subblocks by descending latency.

(3) Schedule:

(a) Single core—in order of sorted list.
(b) Multicore:

(i) Repeat steps (ii) to (viii) until list gets empty.
(ii) temp = top (list) (ready subblock).
(iii) Schedule temp and increment TSct of this core.
(iv) Update TFns of temp and TRdy for all immediate successors.
(v) If any immediate successor of temp is ready (check in order of list) and list is nonempty.
temp = immediate successor.
Go to step (iii).
(vi) If TSct of current core is less than max schedule time and list is nonempty.
Go to step (ii).
(vii) Max schedule time = TSct of current core.
(viii) If list is nonempty switch core.
Go to step (ii).
(ix) END.

4.3. Maximum Dependent Subblock First (MDS). The scheduling decision of MDS algorithm is purely based on the structure of the SDG. The subblock having maximum successors is given higher priority and is picked by the scheduler first to schedule it onto the core with less TSct. The MDS scheduler maintains the ready list. Priority of subblock SB_iB_p in ready list is computed based on TRdy, TFns and its dependencies.

A subblock SB_iB_p can be inserted into the ready list if its dependency list is empty; that is, all the subblocks on which SB_iB_p was dependent have finished their execution.

Algorithm 3 (Maximum Dependent Subblock scheduler).

(1) Repeat until all the subblocks are scheduled.

(2) Collect all the subblocks which are ready for execution.

(3) Find out the priorities for all the subblocks in the ready list.

(4) Schedule:

(4.1) Single core:

(i) Schedule the subblock with highest priority onto the core.
(ii) Update the adjacency matrix.

(4.2) Multicore:

(i) Find the core with less TSct.

(ii) Schedule the subblock with highest priority to the core which is selected.
(iii) Update TFns of the subblock and TRdy of all its immediate successors.
(iv) Update the adjacency matrix.

(5) Go to step (1).

(6) END.

4.4. Longest Latency Subblock First (LLSF). LLSF scheduler is alike the DSB scheduler except the choice made to schedule the successor subblock onto the same core. The only choice of selecting subblock is its schedule latency; the scheduler picks the subblock with the highest schedule latency and schedules it onto the core with lowest TSct. The scheduler uses the sorted list of subblocks. Sorting is based on descending order of scheduling latency. Each node in the list contains subblock and its respective latency.

The LLSF takes scheduling decision based on the information computed in (2), (3), (4), and (5).

Algorithm 4 (Longest Latency Subblock First scheduler).

(1) Find latency.

(2) Sort subblocks by descending latency.

(3) Schedule:

(a) Single core—in order of sorted list.
(b) Multicore:

(i) Repeat steps (ii) to (vii) until list gets empty.
(ii) temp = top (list) (ready subblock).
(iii) Schedule temp and increment TSct of the core.
(iv) Update TFns of temp and TRdy for all immediate successors.
(v) If TSct of current core is less than max schedule time and list is nonempty. Go to step (ii).
(vi) Max schedule time = TSct of current core.
(vii) If list is non-empty switch core. Go to step (ii).
(viii) END.

5. Experimental Framework

Considering a sample multicore architecture, a generic compiler framework has been proposed for exploiting the fine grained parallelism and multiplicity of the component. A sample benchmark program is used for analysis of the performance of the proposed framework. The speed-up, power consumption, performance per power, and communication cost are used as a performance metric. The proposed framework is generic and is independent of the compiler used and architecture.

The experimental setup uses Jackcc, an optimizing *C* compiler that generates assembly language program 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 as each of the cores of multiple core processor is considered to be equivalent to the unicore processor.

5.1. Compiler. The proposed work uses Jackcc compiler [27]. This is an open source compiler developed at the University of Virginia. The basic block in the CFG of Jackcc is called *Arena,* and instruction in the block is called *Quad.* The DAG generated by the front end of the compiler is converted into *quad* intermediate representation, and then these quads are used to construct the basic blocks of CFG. Instructions are in SSA form. The original Jackcc compiler used SAME instruction instead of implementing Φ functions, which instructs the register allocator to place the two live ranges into the same physical register. A SSA conversion module has been integrated in the version of Jackcc which is used in this work [28].

5.2. Architecture. The target architecture model used is a fine grained architecture. This architecture exposes the low level details of hardware to compiler. They implement a minimal set of mechanisms in the hardware and these mechanisms are fully exposed to the software, where software includes both runtime system and compiler. Here the runtime system manages mechanisms historically managed by hardware, and the compiler has the responsibility of managing issues like resource allocation, extracting parallel constructs for different cores, and scheduling. These types of architectures can be seen in some Power4 [29], Cyclops [30], and RAW [31, 32] architecture. The multicore environment has multiple interconnected tiles and on each tile there can be one RISC processor or core. Each core has instruction memory, data memory, PC, functional units, register files, and source clock. FIFO is used for communication. Here the register files are distributed, eliminating the small register name space problem. The core is an example of the grid processor family of designs which are typically composed of an array of homogeneous execution nodes. Gem5 simulator is used to simulate the multicore architecture [33].

5.3. Benchmarks. The test cases that are used to evaluate the proposed work are taken from the Raw benchmark suite [34, 35] and are modified to make it compatible with Jackcc compiler. The raw benchmark suite was designed as part of a RAW project at MIT and is maintained under CVS (Concurrent Versions Systems) to facilitate benchmarking of reconfigurable computing systems.

6. Analysis

The result discussed in this section is based on the simulated model of the target architecture, with dual core and quad core. The results with three active cores on a quad core machine are presented to illustrate the power optimization possibility.

Amdahl's law for multicore architecture proposed by Hill-Marty [36] is used to analyze the performance in terms of

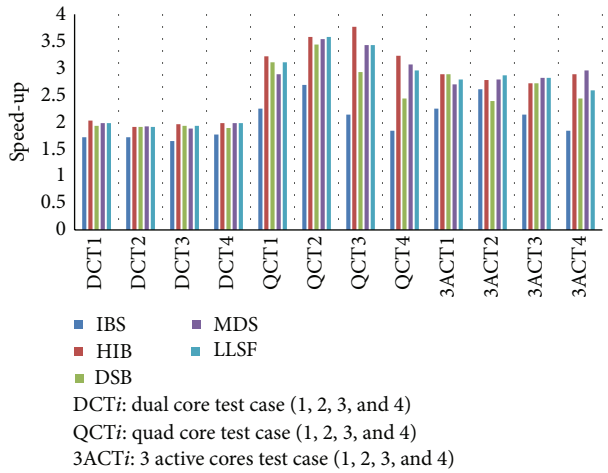


Figure 3: Speed-up analysis.

speed-up. The result is normalized to the performance metric to that of a basic "unit core," which is equivalent to the Base Core Equivalents (BCE) in the Hill-Marty model.

Woo-Lee model [37] 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 consumes in idle state ($0 \leq k \leq 1$). It is assumed that a core in active state consumes a power of 1 unit, that is, the amount of power consumed by one core during the sequential computation phase is 1 unit, while the remaining $(n-1)$ cores consume $(n-1)k$ *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 consumes n *unit* of power. Because it takes $(1-f)$ and f/n to execute the sequential and parallel program, respectively, the formula for average power consumption W in watt given as follows, where f is the fraction of computation that can be parallelized ($0 \leq f \leq 1$):

$$W = \frac{1 + (n-1)\,k\,(1-f)}{(1-f) + f/n}. \tag{6}$$

The model for performance per watt (Perf/W), which represents the performance achievable at the same cooling capacity is based on the average power (W). This metric is essentially the reciprocal of energy because the definition of performance is the reciprocal of execution time. The Perf/W of single-core execution is 1. The Perf/W for multicore is given in

$$\frac{\text{Perf}}{W} = \frac{1}{1 + (n-1)\,k\,(1-f)}. \tag{7}$$

The communication cost is calculated if the dependent subblocks are executed on different cores. The cost of communication depends upon total number of variables a subblock depending on (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 follows:

$$\text{Communication cost is } \begin{cases} \text{Zero:} & \text{if all dependent subblocks are scheduled onto the same core,} \\ \text{Nv} * \text{Ntc} * \text{cf:} & \text{otherwise.} \end{cases} \tag{8}$$

7. Results

The main aim of the proposed work was to equally utilize all the available cores. The result in Figure 3 shows that the speed-up increases as the number of cores increases, which makes it evident that all the cores are utilized towards achieving maximum gain. All the proposed four schedulers contribute to reduced execution time and increase in speed-up. The scheduler with more gain in performance per power is desired during compiling the applications that are used in power critical environment such as mobile embedded devices.

The speed-up decreases when the same program runs on 3 active cores, and at the same time per power performance improves as shown in Figure 5. The power consumed to execute the test cases is captured and performance per power of each test case is calculated as shown in Figures 4 and 5, respectively. It is observed that power increases as the number of cores increases. 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 5. This power optimization can be used in an environment where power is critical. The communication costs of the proposed algorithms are shown in Figure 6 and are compared with the communication cost of the intra block scheduler.

The general observations are as follows.

(i) Speed-up increases as number of cores increase.

(ii) Power consumption increases as the number of cores increases.

(iii) Performance per power decreases when more cores are used.

(iv) Performance per power increases with 3 active cores compared to 4 active cores in a quad core machine with slight compromise on speed-up.

(v) Power increases and speed-up decreases when communication between cores increases.

(vi) Performance per power increases when communication decreases.

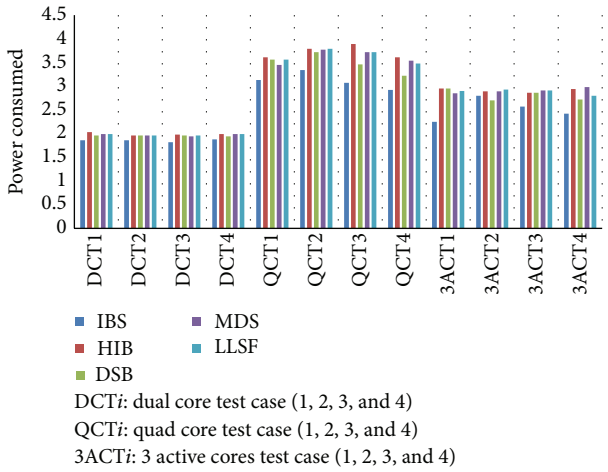


Figure 4: Power analysis.

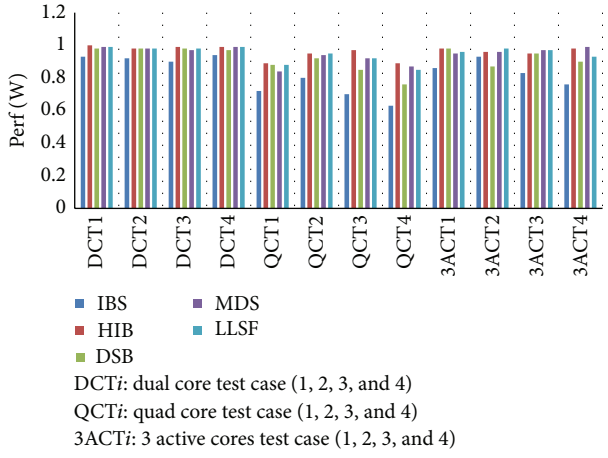


Figure 5: Performance per power analysis.

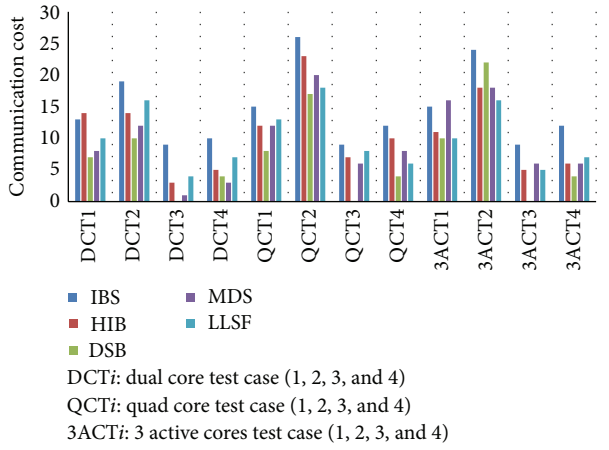


Figure 6: Communication cost.

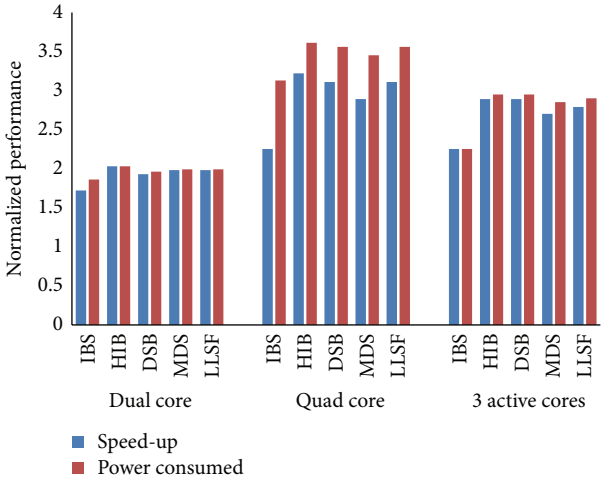


Figure 7: Speed-up versus power consumed.

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

8. Discussion

This section discusses various results obtained using test case 1. In Figure 7, the speed-up and power consumed are compared. Power and performance per power are compared in Figure 8, and the speed-up and communication cost are compared in Figure 9.

The SDG of test case 1 is less dense; that is, the degree of dependency between the subblocks is less. 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 increases in power consumption and decreases the performance per power.

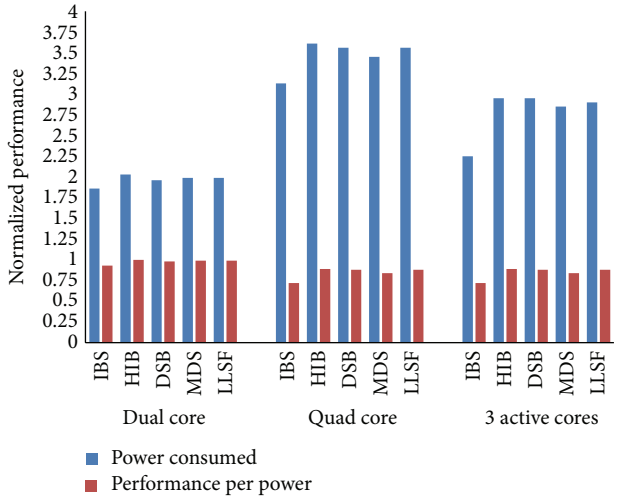


Figure 8: Power versus performance per power.

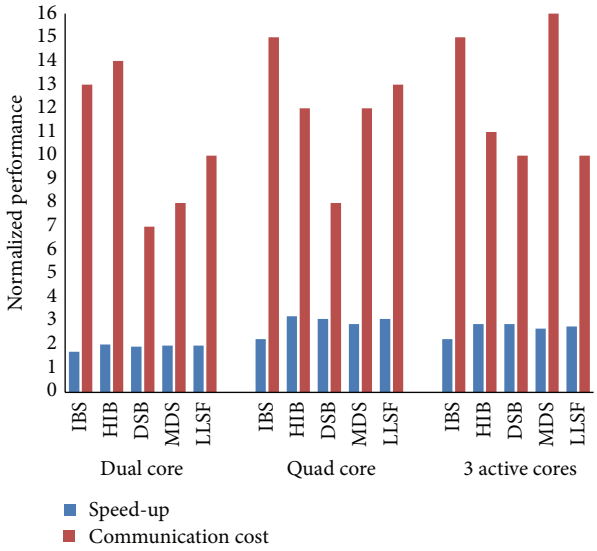


Figure 9: Speed-up versus communication cost.

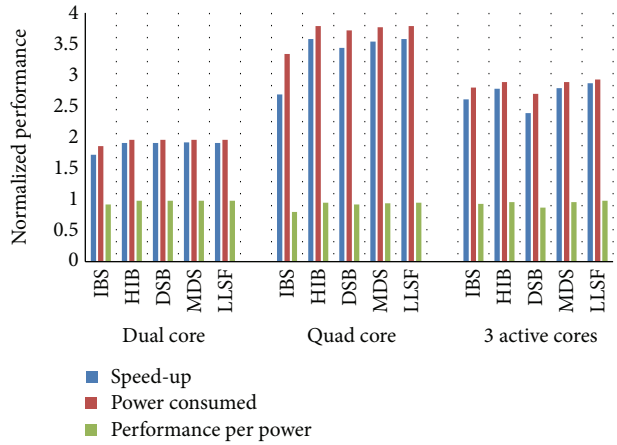


Figure 10: Test case 2.

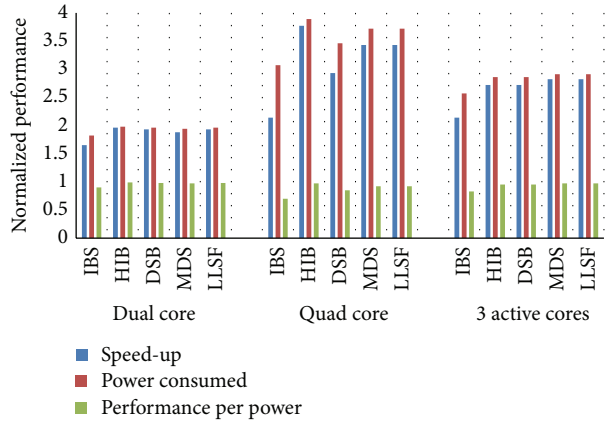


Figure 11: Test case 3.

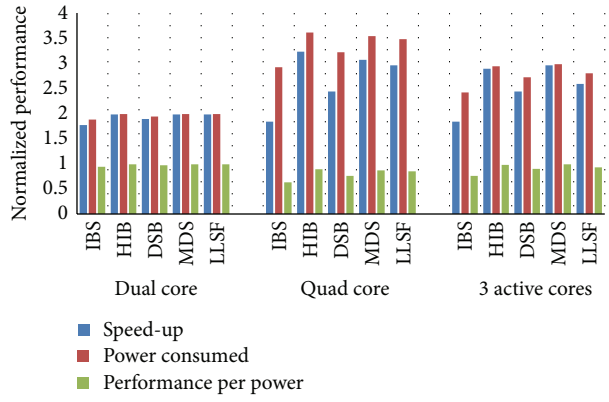


Figure 12: Test case 4.

The dependent subblock based (DSB) scheduler is made memory efficient by scheduling connected subgraphs in SDG, that is, dependent subblocks onto same core. Thus, the DSB scheduler reduces the communication between the cores. But as the number of cores increases this heuristic suffers. This scheduler will not schedule the ready subblock on the other idle cores but would wait to schedule the ready subblock onto the core on which its ancestor subblock 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 subblock 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 achieved using HIB and DSB schedulers. Similar to DSB scheduler, MDS will suffer when SDG is dense and the scheduler does not apply power optimization.

The longest latency subblock first (LLSF) will overcome the limitations of DSB and MDS schedulers. LLSF scheduler picks the subblock with the longest latency and schedules it onto the core with less execution time. The communication cost is less when compared to HIB scheduler but slightly higher than DSB 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 performs linear search to find the subblock with the longest latency.

Similar kind of discussion on relative performance for other three test cases is shown in Figures 10, 11, and 12 and is applicable to other benchmark programs that are used to evaluate the proposed work.

The SDG of test cases 2 and 3 is more dense when compared with SDG of test case 1. The performance per power is better for MDS and LLSF scheduler for test case 2 and test case 3. The HIB scheduler has higher speed-up on these two test cases, but because of more communication among the subblocks scheduled on different cores, the performance per power is reduced when compared with MDS and LLSF scheduler.

The test case 4 is less dense when compared to test case 2 and test case 3. The HIB scheduler is anticipated to outperform, but it induces interleaving in the schedules due to nonavailability of ready subblocks, that is, when TSch of all the subblocks in dependency matrix is greater than TStc of all the cores.

9. Conclusion

The work proposes various compiler level global scheduling techniques for multicore processors. The goal underlying these techniques is to promote the 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 subdivided into the multiple subblocks and thereby a subblock dependency graph is constructed. The proposed schedulers, depending on subblock dependency and their order of execution, allocate the subblocks 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 intercore 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 speed-up 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 subblock formation technique is independent of the number of cores and memory access. Memory contention can have an impact on scalability which would need to be further investigated.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

The authors thank Nick Johnson of University of Virginia for providing the compiler and its assembler.

References

[1] A. Vajda, *Programming Many-Core Chips*, Springer, New York, NY, USA, 2011.

[2] V. S. Pai, P. Ranganathan, and S. V. Adve, "Impact of instruction-level parallelism on multiprocessor performance and simulation methodology," in *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA '97)*, pp. 72–83, February 1997.

[3] S. William, *Computer Organisation and Architecture*, Pearson Education, 8th edition, 2010.

[4] 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, no. 2, pp. 218–226, 1999.

[5] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The performance potential of data dependence speculation & collapsing," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29 '96)*, pp. 238–247, IEEE, Paris, France, December 1996.

[6] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: dynamic optimizations for trace cache microprocessors," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 173–181, December 1998.

[7] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985.

[8] J. B. Dennis and G. R. Gao, "An efficient pipelined dataflow processor architecture," in *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '88)*, pp. 368–373, November 1988.

[9] J. A. Fisher, "The VLIW machine: a multiprocessor for compiling scientific code," *Computer*, vol. 17, no. 7, pp. 45–52, 1984.

[10] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[11] P. Faraboschi, J. A. Fisher, and C. Young, "Instruction scheduling for instruction level parallel processors," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1638–1658, 2001.

[12] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 241–255, Toronto, Canada, June 1991.

[13] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture," *SIGPLAN Notices*, vol. 21, no. 7, pp. 11–16, 1986.

[14] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[15] M. C. Golumbic and V. Rainish, "Instruction scheduling beyond basic blocks," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 93–97, 1990.

[16] P. Robert, P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling computer," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II '87)*, IEEE Computer Society Press, Los Alamitos, Calif, USA, 1987.

[17] M. Lee, P. Tirumalai, and T. Ngai, "Software pipelining and superblock scheduling: compilation techniques for VLIW machines," in *Proceedings of the 26th Hawaii International Conference on System Sciences*, vol. 1, pp. 202–213, Wailea, Hawaii, USA, January 1993.

[18] 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," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[19] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 11, no. 2, pp. 146–178, 1993.

[20] C. Chekuri, R. Motwani, R. Johnson, B. Ramakrishna Rau, and B. Natarajan, "Profile-driven instruction level parallel scheduling," HP Laboratories Technical Report HPL-96-16, 1996.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 2011.

[22] D. C. Kiran, S. Gurunarayanan, and J. P. Misra, "Taming compiler to work with multicore processors," in *Proceedings of the International Conference on Process Automation, Control and Computing (PACC '11)*, pp. 1–6, IEEE, Coimbatore, India, July 2011.

[23] D. C. Kiran, S. Gurunarayanan, F. Khaliq, and A. Nawal, "Compiler efficient and power aware instruction level parallelism for multicore architecture," in *Eco-Friendly Computing and Communication Systems: Proceedings of the International Conference, ICECCS 2012, Kochi, India, August 9–11, 2012*, vol. 305 of *Communications in Computer and Information Science*, pp. 9–17, Springer, Berlin, Germany, 2012.

[24] D. C. Kiran, S. Gurunaraynan, J. P. Misra, and F. Khaliq, "An efficient method to compute static single assignment form for multicore architecture," in *Proceedings of the 1st IEEE International Conference on Recent Advances in Information Technology*, pp. 776–781, March 2012.

[25] D. C. Kiran, B. Radheshyam, S. Gurunarayanan, and J. P. Misra, "Compiler assisted dynamic scheduling for multicore processors," in *Proceedings of the IEEE International Conference on Process Automation, Control and Computing (PACC '11)*, pp. 1–6, IEEE, Coimbatore, India, July 2011.

[26] D. C. Kiran, S. Gurunarayanan, and J. P. Misra, "Compiler driven inter block parallelism for multicore processors," in *Wireless Networks and Computational Intelligence: 6th International Conference on Information Processing, ICIP 2012, Bangalore, India, August 10–12, 2012. Proceedings*, vol. 292 of *Communications in Computer and Information Science*, pp. 426–435, Springer, Berlin, Germany, 2012.

[27] The Jackcc Compiler, http://jackcc.sourceforge.net.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 461–490, 1991.

[29] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[30] C. Cascaval, J. Castanos, M. Denneau et al., "Evaluation of a multithreaded architecture for cellular computing," in *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pp. 311–322, January 2002.

[31] E. Waingold, M. Taylor, D. Srikrishna et al., "Baring it all to software: raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[32] W. Lee, R. Barua, M. Frank et al., "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 46–57, 1998.

[33] The gem5 Simulator System, http://www.gem5.org/.

[34] J. Babb, M. Frank, V. Lee et al., "RAW benchmark suite: computation structures for general purpose computing," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 134–143, April 1997.

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

[36] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[37] D. H. Woo and H.-H. S. Lee, "Extending Amdahl's law for energy-efficient computing in the many-core era," *Computer*, vol. 41, no. 12, pp. 24–31, 2008.