King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

Register allocation for fine grain threads on multicore processor

D.C. Kiran [a],*, S. Gurunarayanan [b], Janardan P. Misra [a], Munish Bhatia [a]

[a] *Department of Computer Science and Information Systems, Birla Institute of Technology and Science Pilani, 333031 Rajasthan, India*
[b] *Department of Electrical Electronics and Instrumentation, Birla Institute of Technology and Science Pilani, 333031 Rajasthan, India*

Received 13 October 2014; revised 3 April 2015; accepted 14 April 2015
Available online 23 November 2015

KEYWORDS

Multicore;
Compiler;
Fine grain parallelism;
Scheduling;
Register allocation

Abstract A multicore processor has multiple processing cores on the same chip. Unicore and multicore processors are architecturally different. Since individual instructions are needed to be scheduled onto one of the available cores, it effectively decreases the number of instructions executed on the individual core of a multicore processor. As each core of a multicore processor has a private register file, it results in reduced register pressure. To effectively utilize the potential benefits of the multicore processor, the sequential program must be split into small parallel regions to be run on different cores, and the register allocation must be done for each of these cores. This article discusses register allocating heuristics for fine grained threads which can be scheduled on multiple cores. Spills are computed and its effect on speed-up, power consumption and performance per power is compared for a RAW benchmark suite.

1. Introduction

Coupled with technological advancement in the field of computer architecture and relentless demand for faster processing has led to the development of multicore processors. A multicore processor has multiple processor cores on same processor chip. Each individual core has a separate register file and is capable of executing complete Instruction Set Architecture (ISA). In order to exploit the capabilities of multicore processors, a 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 the advantage of multiple cores present on the processor chip. To develop an application that exploits multicore, predominantly 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. Currently, several new programing

* Corresponding author.
E-mail addresses: dck@pilani.bits-pilani.ac.in (D.C. Kiran), sguru@pilan.bits-pilani.ac.in (S. Gurunarayanan), jpm@pilan.bits-pilani.ac.in (J.P. Misra), munish.bhatia.cse@gmail.com (M. Bhatia).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2015.04.001
1319-1578 © 2016 Production and hosting by Elsevier B.V. on behalf of King Saud University.

models and different ways to exploit threads and data-level parallelism are being explored which help in coarse grained parallelism. There is very little effort from the research community toward the exploitation of compiler driven fine grained parallelism in 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 (Zhong, 2008). The architecture can be designed to support the minimal set of operations required for executing an instruction and the task of extracting the fine grained parallelism can be left for compilers and run time environment to achieve. The runtime environment can manage resource allocation, extracting parallel constructs for different cores, and scheduling based on information generated by the compiler. Some of the architectures supporting these features are Power4 (Tendler et al., 2002), Cyclops (Cascaval et al., 2002) and RAW (Waingold et al., 1997) architecture. The multicore environment has multiple interconnected tiles and on each tile there can be one RISC like processor or core. Each core has instruction memory, data memory, PC, functional units, register files, and source clock. FIFO (queue) is used for communication. Here the register files are distributed, eliminating the small register name space problem. 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. There has been considerable focus on improving performance through automated fine-grain parallelization, where a sequential program is split into parallel fine grained threads and are scheduled onto multiple cores (Kiran et al., 2011a,b; Kiran et al., 2012). In general, the multicore processors have a private register file, L1 data and Instruction cache and shared L2 cache. The limited size of L1 data cache, warrants the optimal amount of data to be brought into the cache. The poor choices in the placement of data can lead to the increased memory stalls and low resource utilization. The fine grained threads that are scheduled onto different cores need to be allocated registers from a respective register file of the core on which they are scheduled.

Various register allocation approaches are proposed in the past (Chaitin, 1982; Norris and Pollock, 1994; Gupta et al., 1994; Callahan and Koblenz, 1991; Lueh et al., 2000; Chow and Hennessy, 1984; Briggs et al., 1989; Poletto and Sarkar, 1999; Mossenbock and Pfeiffer, 2002; Fu and Wilk, 2002; Burkard et al., 1984; Todd et al., 1996). Most of the register allocation algorithms assume that the CPU has regular register file and these algorithms fail to adapt 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 (Koes and Goldstein, 2005; Kong and Wilken, 1998; Scholz and Eckstein, 2002). In the case of a multicore processor, each core of the processor has an individual register file and optimal register allocation is of utmost importance. Multicore architecture is one area which expects new thinking for register allocation. The proposed work explores the various likely steps needed for register allocation for multicore architecture.

This paper proposes two register allocation heuristics referred as heuristic 3 and heuristic 4 for fine grained threads which can be scheduled on multicore processor. Results are compared with heuristic 1 and heuristic 2 which are existing register allocation approaches. The proposed register allocation heuristics along with considering multiple private register files on each core, constructs the interference graph incrementally by checking the register pressure as opposed to existing register allocation approaches which construct the global interference graph and then perform simplification to reduce register pressure.

The rest of the paper is organized as follows. Section 2 describes background of the proposed work. It also discusses some of the recent works pertaining to sub-block creation or fine grained thread extraction and scheduling techniques. Section 3 gives a detailed description of the proposed register allocation technique for multicore environment. Through an illustrative example the steps involved in the proposed algorithm is presented in Section 3.4. Analysis and discussion of the results are presented in Section 4, Section 5 presents the conclusion and direction for future work.

2. Background

This section introduces the background of the proposed work. The proposed work performed in conjunction with following work.

- Parallel region formation or extracting fine grain threads (Kiran et al., 2011a).
- Scheduling parallel regions or fine grain threads on to multiple cores (Kiran et al., 2011b, 2012).

The work flow of the compiler is shown in Fig. 1. Two additional passes are introduced into the normal flow of the compiler. The Fine grained extractor module and the scheduler module. The Fine grained extractor module analyzes the basic blocks of CFG and divides each of them into multiple sub-blocks. The scheduler module generates the multiple schedules which can be concurrently executed on different cores of the processor. The register allocator module carries out the register assignment operation using Chaitin's register allocation approach (Chaitin, 1982).

2.1. Fine grain thread

A fine grain thread is a sub-block formed by analyzing the instruction dependency in the basic block of the control flow graph (CFG) of a program Kiran et al., 2011a. The fine grain thread extractor module in Fig. 1 creates sub-blocks. The sub-blocks created are disjoint and can run in parallel. In Fig. 3, the CFG has 4 basic blocks (B_p). The disjoint set operations

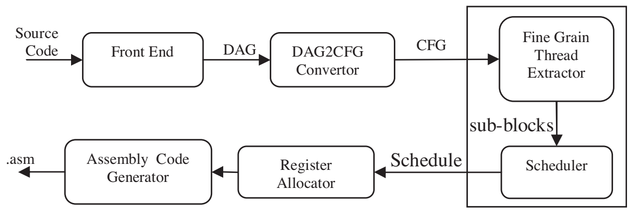


Figure 1 Flow of compiler.

are applied to 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.

2.2. Sub-block dependency graph (SDG)

The SDG is a graph $G(V, E)$, where every vertex $v \in V$ is a sub-block denoted as SB_iB_p. An edge $e \in E$, is added between two dependent vertices $SB_i \in B_p$ and $SB_j \in B_q$ where $p \neq q$. In Fig. 3(a), sub-block SB_1 belonging to basic block B_2 is dependent on sub-block SB_1 belonging to basic block B_1 then it is denoted as $SB_1B_1 \rightarrow SB_1B_2$ and the sub-block SB_1B_2 should be scheduled only after sub-block SB_1B_1 completes its execution.

2.3. Scheduler

The scheduler (Inter block scheduler) identifies all the independent sub-blocks in a CFG and formulates the schedule (Kiran et al., 2012). The scheduler tries to generate schedules for each core. Each schedule consists of a list of sub-blocks that can be scheduled on a core.

In general the global scheduler selects the sub-block SB_iB_p from the sub-block dependency matrix if its dependency list is empty. Sub-block dependency matrix is a jagged matrix representation of SDG as shown in Fig. 3(b) where the first column contains a list of all sub-blocks in SDG and all sub-blocks on which a specific sub-block is dependent are listed in the corresponding row. Once SB_iB_p is scheduled and completes its execution, its corresponding entries are removed from the dependency list. The sub-block 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 amounts of time, without loss of generality, it is assumed that the average time taken to execute an instruction is constant.

The decision of scheduling a sub-block on a core is based on the invariants such as scheduling latency, computed ready time (T_{rdy}) and finish time (T_{fns}) of the sub-block SB_iB_p. The current schedule time (T_{sch}) and available time (T_{ca}) of the core are also taken into consideration to check the availability of a core to schedule the sub-blocks.

Ready_time (T_{rdy}) 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.

Finish_time (T_{fns}) of a sub-block is a summation of starting time interval when the sub-block is scheduled and total time taken to complete the execution.

Schedule time (T_{sch}) is the time when a ready sub-block is likely to be scheduled on the core. It is computed by finding the maximum of the finish time of the currently executing sub-block, if any, and T_{rdy} of the sub-block to be scheduled.

Core available time (T_{ca}) for a given core is the finish time of the currently executing sub-block on that core.

The sub-blocks with empty dependency lists are moved to the read list, which is arranged in descending order of scheduling latency of sub-blocks. The ready sub-block is scheduled on the core with least T_{ca} value. To compute the scheduling latency of a given sub-block SB_iB_p in SDG, all the paths leading to leaf node are identified. Each path starting from the sub-block SB_iB_p to a leaf node can be considered as a linear list. The scheduling latency for each path is computed and the sub-block SB_iB_p from the path with high latency is chosen for scheduling. The scheduling latency is a summation of execution time of all the nodes in the linear list, which can computed by summing up the number of instructions in each node in the list. It is assumed that execution time is proportional to the number of instructions. The scheduling latency of leaf sub-block SB_iB_p in SDG is, the total number of instructions in that sub-block. The latency of the non-leaf sub-block SB_iB_p is the summation of maximum latency of all its immediate successor, sub-blocks and total number of instructions in SB_iB_p. The Fig. 4a shows the generation of two schedules for a dual core processor.

3. Register allocation for multicore processor

This section describes the proposed register allocation framework as shown in Fig. 2.

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.

Initial interference graphs are built locally for each sub-block listed in the sub-block list. These interference graphs are mostly k-colorable, if not they are simplified and the variables which are to be spilled are captured. The spill code is inserted after register assignment phase.

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 a merge operator. The merge operation is built using two functional modules, mergeSubblocks and checkSimplifiable and produces a list of hyper sub-blocks H (h1, h2, h3...hx) whose interference graph is k-colorable. Hyper sub-blocks ensure temporal locality by pushing maximum dependent instructions on the core for execution. As the hyper sub-blocks are k-colorable, zero spilling is required and instructions will remain inside private memory of individual cores till all the instruction commits without doing external memory reference. The variables in the hyper sub-block are assigned register at the register assignment phase.

3.1. Merge operator

The merge operator produces k-colorable hyper sub-blocks by merging the interference graphs of sub-blocks listed in the schedule list. 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.

The conditions used in Algorithm 1 are listed below.

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: Do not merge the sub-blocks.
D2: Merge the sub-blocks to schedule and allocate registers together.

The algorithm begins by selecting two sub-blocks SB_i and SB_j which is followed by a dependency constraint check. The constraints are enforced through the condition C1, C2 and C3. These conditions are derived from the invariants used by the global scheduler.

Assuming that SB_j is listed in the schedule of the 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 a hyper sub-block.

The sub-block SB_j can be merged with its predecessor SB_i if it is not dependent on sub-block/s SB_k and 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 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 3.4.

Algorithm 1. 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
  else if(!C1)
  begin
    if(C3 & C4)
    begin
      Merge the sub-blocks to schedule and allocate register.
      end

  end

  else
  begin
    Do not merge the sub-blocks.
  end
end
```

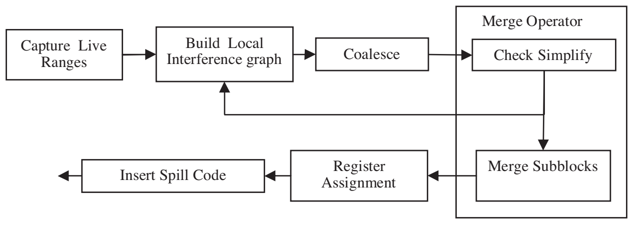


Figure 2 Modified register allocation framework.

The disjoint-set forests (Cormen et al., 2001) 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.

3.2. 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 a spill code is eliminated. The choice of the order of coloring is simplified due to the fact that the interference graph is Chordal with simplicial vertex (Pereira and Palsberg, 2005). The edge projecting out of the simplicial vertex is pushed onto the color stack first and continued till all the edges are pushed onto the stack. Once all the edges are pushed onto the stack, the color assignment module pops out the edges from the stack to assign different colors 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.

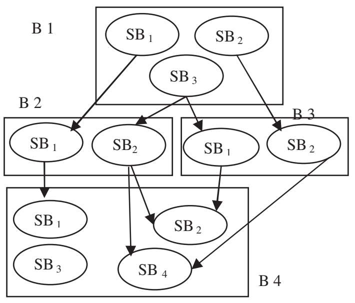
3.3. Insert spill code

In this phase, the spill code load/store is inserted for the spilled variable which is captured during the construction of initial interference graphs of the sub-blocks. However the interference graphs of the hyper sub-blocks are *k*-colorable which eliminate the need of the spill code. Spill codes are inserted after the creation of hyper sub-blocks and register assignment phase to retain the properties of SSA (Cytron et al., 1991; Hack and Goos, 2006; Pereira and Palsberg, 2006).

3.4. Working example

This section using an example illustrates the proposed register allocation approach for the cores having four registers each. To extract fine grained parallel threads, the disjoint sub-blocks and SDG is created and is shown in Fig. 3(a).

The schedule created by the global scheduler for dual core machine is shown in Fig. 4(a). The hyper sub-block generated by using the schedule list is shown in Fig. 4(b). Fig. 4(c) exposes the instructions in sub-blocks SB_2B_1 and SB_2B_3 scheduled on core 2 of dual core machine. The sub-block SB_2B_1 and its successor SB_2B_3 are scheduled on core 2 as shown in Fig. 4 (a). The sub-block SB_2B_3 does not satisfy the condition C1 as it is not dependent on any other sub-block. The condition C3 is



	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 3 (a) Sub-block dependency graph, (b) sub-block dependency matrix.

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)

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

(c)

Figure 4 (a) Sub-block lists generated for dual core processor, (b) hyper sub-blocks whose interference graph is *k*-colorable, (c) instructions in sub-blocks SB_2B_1 and SB_2B_3.

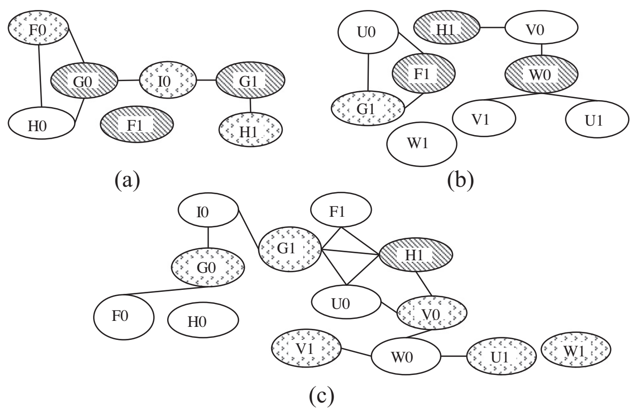


Figure 5 (a) Interference graph of sub-block $SB_2B_{1,}$ (b) interference graph of sub-block $SB_2B_{3,}$ (c) interference graph of merged sub-blocks.

also not satisfied as no other sub-blocks are dependent on SB_2B_1. It is evident from the Fig. 5(a) and (b) that the simplifiable condition C4 is satisfied for the interference graphs of sub-blocks SB_2B_1 and SB_2B_3. Further Fig. 5(c) shows that the merged interference graph of sub-blocks SB_2B_1 and SB_2B_3 is also simplifiable (3 colorable) therefore the sub-blocks SB_2B_1 and SB_2B_3 are merged to form a single hyper sub-block. An attempt to schedule the sub-block SB_2B_3 by merging with its successor SB_2B_2 into the hyper sub-block fails. The condition C1 is satisfied as the sub-block SB_2B_2 is dependent upon the sub-block $SB_3B_{1.}$ The condition C2 is also not satisfied because the T_{fns} of the sub-block on which SB_2B_2 depends, i.e. the sub-block SB_3B_1 is not less than T_{rdy} of $SB_2B_{3.}$ Thus the decision D1 is taken not to merge the sub-block SB_2B_2 into the hyper sub-block.

4. Experiments

This section is divided into five sub sections. Section 4.1 details the architecture, compiler and benchmarks used for the experiment. Section 4.2 gives detailed evaluation criteria, for different register allocation heuristics. A brief explanation of different register allocation heuristics is given in Section 4.3. Section 4.4 discuses the compilation cost in terms of algorithmic complexity of the register allocation approach. Table 2 in the sub Section 4.5 gives the spills of different heuristics and Figs. 6 and 7 show the effect of spilling on speed-up, power and performance per power when a program is compiled for dual core and quad core processors respectively.

4.1. Multicore architecture

The target architecture model used is capable of executing fine grained threads. The multicore processor is an example of the grid processor family which is typically composed of an array of homogeneous execution nodes. The core is equivalent to a single threaded uni-processor capable of executing 1 instruction per cycle and each core has 8 general purpose registers

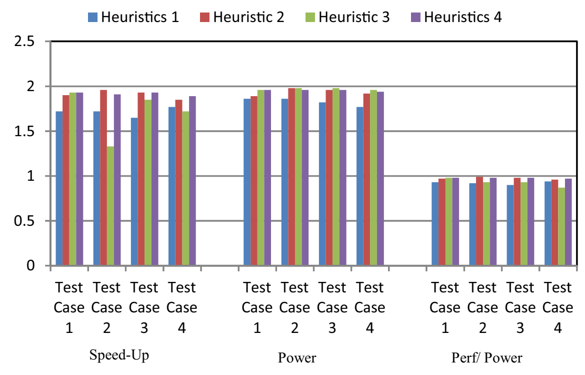


Figure 6 Speed-up, power and perf/power comparison on dual core machine.

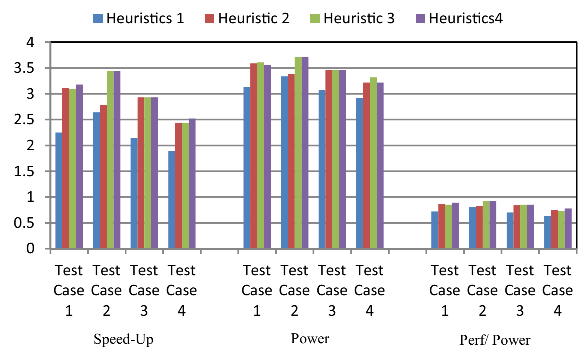


Figure 7 Speed-up, power and perf/power comparison on quad core machine.

[\$r8....\$r15]. Each core has instruction memory, data memory, PC, functional units, register files, and a source clock. FIFO is used for inter core (threads executing on different core) communication. Here the register files are distributed, eliminating the small register name space problem. The proposed work uses open source Jackcc compiler (Jack, 2013). The test cases to evaluate the proposed work are taken from the Raw benchmark suite (Babb et al., 1997).

The target architecture is an irregular architecture as the registers files are distributed. The principle behind this design is to avoid unnecessary, redundant, and complex connections with other functional units, thus lowering the large area, increasing the access speed, and lowering power consumption.

4.2. Evaluation criteria

The results discussed in this section are based on the simulated model of the target architecture, with Dual core and Quad core variants. The result of four extreme test cases of the benchmark suite such as DES, Integer matrix multiplication, Fast Fourier transform and Merge sort are presented by analyzing and comparing the spill cost, the speed-up, power consumption, and performance per power. The result is normalized to the performance metric to that of a basic "unit core", which is equivalent to the "Base Core Equivalent (BCE)" in the Hill-Marty model (Hill and Marty, 2008; Woo and Lee, 2008). The global scheduler is designed to perform power optimization, i.e. if the performance achieved using four cores can be achieved using only three cores, then only three cores of the quad core machine are used to execute the given task, either by keeping the 4th core idle or utilizing it for some other computation (Kiran et al., 2012). The results with three active cores on a quad core machine are presented to illustrate the power optimization possibility. 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.

4.3. Register allocation heuristics

This section describes different register allocation heuristics for multicore processor architecture whose results are compared. The first heuristic uses the raw list 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 integrate register allocation with global scheduling (Kiran et al., 2013). It mainly aims at addressing the phase ordering problem which leads to poor optimization. Taking the register requirement into the account the scheduler creates schedules for multiple cores by selecting sub-blocks in the dependency matrix to maintain the order of their execution.

Heuristic 3 and heuristic 4, follow the proposed register allocation framework as shown in Fig. 4. To facilitate the global scheduling, these heuristics incrementally merge the sub-blocks in the sub-block list generated by scheduler to produce a hyper sub-block list. The hyper sub-blocks ensures temporal locality by pushing maximum dependent instructions on to core for execution. These heuristics help to produce the optimized code at the cost of increased compilation time.

Table 1 Algorithm complexity comparison.

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

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 (T_{rdy}) and Finish_time (T_{fns}) of the sub-blocks that are being merged. In these 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 the spill code is inserted. Since each hyper sub-block can be assigned a thread, it improves the execution time as compared to heuristics 1 but may add more spill code as an interference graph may not be k-colorable.

The proposed fourth heuristic overcomes the limitations of heuristic 1 & 3. In this approach the hyper sub-blocks are created by adding a 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.

4.4. Algorithm complexity

In this section complexity analysis for the proposed register allocation heuristic is presented. The complexity of other three heuristics are compared and presented in Table 1.

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($m * n$ log$n$). 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 in 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($m * n^2$).

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

4.5. Result and analysis

- Heuristic 1 does not contribute a 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.

Table 2 Spills.

Algorithm		Heuristics 1,4	Heuristic 2	Heuristic 3
Test case 1	Dual core	0	1	0
	Quad core	0	1	1
	3-active cores	0		1
Test case 2	Dual core	0	1	3
	Quad core	0	1	0
	3-active cores	0		0
Test case 3	Dual core	0	0	3
	Quad core	0	0	0
	3-active cores	0		0
Test case 4	Dual core	0	1	2
	Quad core	0	0	2
	3-active cores	0		2

- Heuristic 2 tries to solve the phase ordering problem by allocating register during scheduling by compromising a little with the performance. This heuristic contributes a 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 contributes a 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 4th heuristics are used is shown in Table 1. The spilling is almost zero when register allocation is done on the list of sub-block using heuristic 1 as the sub-blocks are created by taking register requirement into account. Similarly, as the interference graph of hyper sub-blocks is are k-colorable the proposed heuristic 4 produces zero spill.
- As heuristic 2 is not designed for power optimization, the results for 3-active cores in Table 2 are left blank.

The performance gain is a combined effort of scheduler and register allocation approach. Results in Figs. 6 and 7 depict the effect of spilling on speed up, power and perf/power of dual core and quad core processor.

- The speed-up and performance per power of heuristic 1 is lower even with zero spills. This is because of the limitation of the local scheduler. The local scheduler assigns a 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 deteriorates for test case 2 on a 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 cases 2, 3 and 4 to execute on the dual core machine, the speed-up decreases due to spilling. This results in higher power consumption and reduced performance per power in comparison to heuristic 4.
- Test case 3 on a 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 cases on a quad core processor.
- Heuristic 4 also offers better performance in respect to compilation time.
- It is evident from the Figs. 6 and 7 that the overall performance of heuristic 4 is better with respect to performance per power in comparison to heuristics 1, 2 and 3.

4.6. *Observation on number of registers*

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

Interference graphs are built incrementally by checking dependency and simplifiablity conditions. Increasing the number of registers can cause an 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 a larger hyper sub- block will require less data movement to and from memory.

According to the chromatic polynomial theory for the chordal graph, a chordal graph with a largest clique of size n will have chromatic number n. The largest clique in interference graph of the SSA form program is 3, thus 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 patterns for $k + 1 \leqslant 8$ where k ranges 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 the 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.

5. Conclusion

The work proposes a register allocation mechanism to be used for multicore processors. 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 unicore processors. The experimental results presented in the paper 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 for the code generated for the programs from the RAW benchmark suite. The use of this register allocation technique for multicore processors is more efficient than the use of conventional register allocation approach.

The proposed work can be extended for the multicore processors with each core having multiple execution units (floating point unit, integer unit) (Bhathia et al., 2013). Though the register files are not distributed based on types (floating point registers, integer registers) the register requirement and the proportion of the type of instruction can be considered while performing register assignment.

References

Babb, J., Frank, M., Lee, V., Waingold, E., Barua, R., Taylor J. Kim, M., Devabhaktuni, S., Agarwal, A., 1997. The RAW benchmark suite: computation structures for general purpose computing. In: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, p. 134.

Bhathia, Munish, Kiran, D.C., Gurunarayanan, S., Misra, J.P., 2013. Fine Grain Thread Scheduling on Multicore Processors: Cores with Multiple Functional Units. ACM Compute.

Briggs, P., Cooper, K.D., Kennedy, K., Torczon, L., 1989. Coloring heuristics for register allocation. In: Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 275–284.

Burkard, Rainer E, ela, Eranda C, Pardalos, Panos M, Pitsoulis, Leonidas S, 1984. Quadratic assignment problems. European J. Oper. Res. 15, 283–289.

Callahan, David, Koblenz, Brian, 1991. 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.

Cascaval, C., Castanos, J., Ceze, L., Denneau, M., Gupta, M., Lieber, D., Moreira, J.E., Strauss, K., Warren, H.S., Jr. 2002. Evaluation of multithreaded architecture for cellular computing. In: Proceedings of the 8th International Symposium on High Performance Computer Architecture, pp. 311–322.

Chaitin, G., 1982. Register allocation and spilling via graph coloring. In: Proceedings of the SIGPLAN Symposium on Compiler Construction, pp. 98–105.

Chow, Fredrick, Hennessy, John, 1984. Register Allocation by priority-based coloring. In: Proceedings of the ACM SIGPLAN Symposium on Compiler Construction SIGPLAN Notices, 19(6).

Cormen, T., Leiserson, C., Rivest, R., 2001. Introduction to Algorithms. The MIT Press, Cambridge, MA.

Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., 1991. Efficient computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13 (4), 451–490.

Fu, Changqing, Wilk, Kent, 2002. A faster optimal register allocator. In: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35).

Gupta, Rajiv, Lou Soffa, Mary, Ombres, Denise, 1994. Efficient register allocation via coloring using clique separators. ACM Trans. Program. Lang. Syst. 16 (3), 370–386.

Hack, S., Goos, G., 2006. Optimal register allocation for SSA-form programs in polynomial time. Inf. Process. Lett. 98 (4), 150–155.

Hill, M.D., Marty, M.R., 2008. Amdahl's Law in the multicore era. IEEE Comput., 33–38

Woo, Dong Hyuk, Lee, Hsien-hsin S., 2008. Extending amdahl's law for energy-efficient computing in the many-core era. IEEE Comput., 24–31

Kiran, D.C., Gurunarayanan, S., Misra, J.P., 2011a. Taming compiler to work with multicore processors. IEEE Conf. Process Autom. Control Comput.

Kiran, D.C., Radheshyam, B., Gurunarayanan, S., Misra, J.P., 2011b. Compiler assisted dynamic scheduling for multicore processors. IEEE Conf. Process Autom. Control Comput.

Kiran, D.C., Gurunarayanan, S., Faizan, Khaliq, Nawal, Abhijeet, 2012. Compiler efficient and power aware instruction level parallelism for multicore architectures. The International Eco-friendly Computing and Communication Systems, Published in the Communications in Computer and Information Science (CCIS). Springer-Verlag, pp. 9–17.

Kiran, D.C., Gurunarayanan, S., Misra, J.P., 2012. 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.

Kiran, D.C., Gurunarayanan, S., Misra, J.P., Yashas, D., 2013. Integrated scheduling and register allocation for multicore architecture. In: IEEE Conference on Parallel Computing Technologies PARCOMPTECH-2013, Organized by C-DAC in IISC Bangalore.

Koes David, Goldstein, Seth Copen, 2005. A progressive register allocator for irregular architectures. In: CGO, pp. 269–280.

Kong, Timothy, Wilken, Kent D., 1998. Precise register allocation for irregular architectures. International Symposium on Microarchitecture. ACM, pp. 297–307.

Lueh, Guei-Yuan, Gross, Thomas, Adl-Tabatabai, Ali-Reza, 2000. Fusion-based register allocation. ACM Trans. Program. Lang. Syst. 22 (3), 431–470.

Mossenbock, Hanspeter, Pfeiffer, Michael, 2002. Linear scan register allocation in the context of SSA form and register constraints. In: CC, LNCS, pp. 229–246.

Norris, Cindy, Pollock, Lori L., 1994. Register allocation over the program dependence graph. SIGPLAN 94-6/94.

Pereira, Fernando Magno Quintao, Palsberg, Jens, 2005. Register Allocation via Coloring of Chordal Graphs. In APLAS, Springer, pp. 315–329.

Pereira, Fernando Magno Quintao, Palsberg, Jens, 2006. Register allocation after classic SSA elimination is np-complete. In: Foundations of Software Science and Computation Structures. Springer.

Poletto, Massimiliano, Sarkar, Vivek, 1999. Global linear scan register allocation. ACM Trans. Program. Lang. Syst. 21 (5), 895–913.

Todd, A., Proebsting, Fischer, Charles N., 1996. Probabilistic register allocation, ACM SIGPLAN '92 PLD1-6/92/CA.

Scholz, Bernhard, Eckstein, Erik, 2002. Register allocation for irregular architectures. LCTES/SCOPES. ACM, pp. 139–148.

The Jack Compiler, http://jackcc.sourceforge.net.

Tendler, J.M., Dodson, J.S., Fields, J.J.S., Le, H., Sinharoy, B., 2002. Power 4 system microarchitecture. IBM J. Res. Dev. 46 (1), 5–6.

Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A., 1997. Baring it all to software: raw machines. Computers 30 (9), 86–93.

Zhong, Hongtao, 2008. Architectural and Compiler Mechanisms for Accelerating Single Thread Applications on Multicore Processors (Ph.D. Dissertation). University of Michigan, Ann Arbor (ACM), MI, USA.