Critical Path Heuristic for Automatic Parallelization

Mirza Omer Beg
David R. Cheriton School of Computer Science
University of Waterloo
Technical Report CS-2008-16

August 6, 2008

Abstract

This paper presents an algorithm to heuristically partition the data dependence graph representing a basic block or a hyper block of code in order to assign instructions to available processing cores. The algorithm relies on the observation that the critical path in the data dependence graph defines the lower bound on the schedule length of a block and hence the instructions on the critical path should be assigned to the same core. The algorithm has been implemented in the Trimaran compiler framework. The given heuristic algorithm is much simpler and evaluation shows that it is as good as previous state-of-the-art heuristic algorithms.

1 Introduction

Instruction scheduling techniques have traditionally targeted straight-line code in the form of *basic blocks*, *super blocks* and *hyper blocks* to generate schedules for exploiting instruction-level parallelism. A basic block is a sequence of instructions with a single entry point called *source* and a single exit point called *sink*. Successive basic blocks which result in a straight-line region of code with a single entry point and multiple exits form a super block. A hyper block is a super block where the control dependencies are converted into data dependencies using special control registers. Recent trends in microprocessor design, with multiple cores per chip, necessitates changes in compiler design in order to make better use of available hardware parallelism.

Multi-processor machines have been around for over three decades. But, only in the last few years has the technology migrated into the mainstream as the new generation of processors, namely *multicore* [3, 15]. Today's multicores are very much like the multi-processor systems of yesterday. Multicores have merely clustered the processors (referred to as *cores*) as well as a subset of the memory hierarchy onto a single chip, reducing the cost of communication between cores and making way for cost-effective parallel execution of program threads.

Parallelization is the technique which converts sequential code into multi-threaded code in order to gain performance from available hardware parallelism. As multicore architectures become mainstream, parallelization of sequential code is regaining the attention of the compiler research community. Software developers can no longer rely on increasing clockspeeds for performance improvements of single threaded applications. This is mainly because the recent trends in architecture design resulting in increasing number of cores per chip coupled with little or no improvements in clockspeeds per core. This poses a tremendous challenge for compiler designers and necessitates

automated parallelization to evenly distribute the workload on the available processor cores. This would be complementary to traditional parallelism extraction techniques such as vectorization which look for coarser level parallelism. The increasing number of cores per chip also means that effective parallelization techniques can result in significant speedups for general purpose applications. This is evident from performance gains achieved by other complementary, semi-automated and user-assisted parallelization techniques incorporated in the RapidMind platform [14] and OpenMP [5].

There is a small body of work in literature that also makes an attempt at heuristically partitioning straight-line regions of code on multi-processor architectures [6, 9, 12, 13, 16]. This work targets clustered architectures and predates the multicore.

The rest of the paper is organized as follows. An overview of the required background material is given in the next section. Section 3 gives details of the heuristic algorithm. Section 4 describes the experimental setup and results. Section 5 gives an overview of the related work. Section 6 gives a discussion and analyzes the approach given in this paper. Finally, the paper concludes with Section 7.

2 Background

This section provides the necessary background required to understand the approach described in the rest of the paper. It also gives an introduction to the problem that this paper attempts to solve along with the assumptions and the architectural model.

2.1 Problem Statement

The problem of assigning and scheduling instructions for multicores can be defined in terms of *schedule length*.

Definition 2.1 (Schedule Length)**.** Given the dependence graph for a code block the schedule length is the cycle in which the last instruction is issued.

Given the definition of the schedule length, which applies to basic blocks as well as hyper blocks, the problem can be stated as follows.

Definition 2.2 (Assignment for Multicores)**.** Given the dependence graph $G = (V, E)$ for a code block and the number of available cores k, the problem is to find an assignment $A(i) \in \{1, \cdots, k\}$ to all the instructions $i \in V$ that minimizes the schedule length of the block.

With the above definition the problem can be restated as a graph partitioning problem.

Definition 2.3 (Partitioning)**.** Given the dependence graph for a code block and k cores partition the dependence graph into k or fewer subgraphs such that the difference between gain from parallelization and cost of synchronization is maximized.

Balanced graph partitioning is an NP-hard problem [2]. The partitioning problem defined above is harder than simple balanced partitioning as the feasible partitions may be fewer than k. This paper presents a heuristic technique to solve the partitioning problem. Figure 1 shows a data dependence graph of a basic block and how it can be partitioned for $k = 2$.

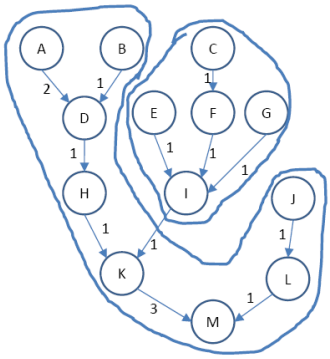

Figure 1: Partitions of an example basic block.

2.2 Theoretical Limits

It is important to understand that any parallelization technique will be bounded by theoretical limits. Gene Amdahl predicted that the maximum achievable speedup from parallelization is limited by the percentage of non-parallelizable portion of a program [1]. This is known as Amdahl's law. The law states that if the sequential (non-parallelizable) portion of the program is α the maximum speedup achievable on a system with k processors can be given by

$$S(k) = \frac{k}{1 + \alpha * (k - 1)}$$

Theoretically, as k tends to infinity, the maximum speedup tends to $1/\alpha$. Assuming fixed sized programs, the law implies that influence of the sequential portion of the program increases with the number of processors.

2.3 Architectures and Our Model

Modern multicores differ a lot in their designs. Some multicore chips have multiple complete processors (with their own separate caches) placed on the same die (e.g. dual-core AMD Opteron) whereas others share the cache at some level on the chip (e.g. Intel Xeon Woodcrest). The functional units and the architectural state is completely independent but the cores share a common interface to system memory and input/output devices.

The cores access a shared memory space and coherence is maintained between the private caches (e.g. the L1) using a coherence protocol. The caches are all physically addressed. Thus the threads can fully share the cache contents. Multiple threads communicate with each other through the memory hierarchy. The producer thread writes the data to an address and the consumer thread reads from the same memory address. Since the cores are on the same die the communication latency is much smaller than the traditional multi-processor systems. For example in the Intel Core Duo processor, which has a shared on-chip L2 cache, the communication latency between cores can be as small as 14 cycles.

For the purpose of this project a simple multicore architectural model is assumed. Multicore processors can have a number of identical processing cores integrated onto a single chip. In general,

the following holds for our architecture model.

- Cores are homogeneous. This means that all cores have the same number of identical functional units.
- A perfect memory model, meaning that memory load/store latencies are ignored in this model.
- Let $l(i, j)$ be the latency between the cycle the instruction i is issued on a core and when the result is available to be used by instruction j.
- Each core is single issue. This implies that only one instruction can be issued at any given cycle on any particular core.
- Let some non-zero s be the cost of synchronization between two cores. After the result of an instruction is available, it would take s cycles to transfer the resultant value on a different core where it is needed.

3 Critical Path Heuristic

3.1 Critical Path

The *critical path* in a dependence graph is the set of edges that have the greatest sum of latencies from a source node to a sink node. For example Figure 2 shows the critical path of the basic block dependence graph shown in Figure 1.

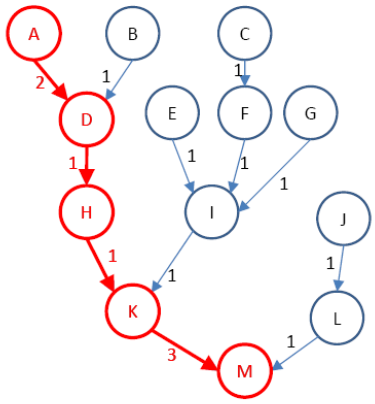

Figure 2: Critical path in the example basic block.

Since the instructions on the critical path have to be execute and no instruction can be executed before its successor in the dependence graph, the lower bound for the schedule length of a basic block would at least be the length of the critical path regardless of how many processors are available.

$$Schedule\ Length \geq \sum_{(i,j) \in cp(\mathcal{G})} l(i, j) \tag{1}$$

The heuristic algorithm is based on the idea that all instructions on the critical path be assigned to the same processor core. Applying Amdahl's law to the problem, the heuristic assumes that the critical path in a code block is the non-parallelizable portion of that block and the maximum speedup is limited by the length of this critical path compared to the total number of nodes in the block.

3.2 Weight Assignment

3.2.1 Node and Edge Weights

Since the architecture of each core assumed in this paper to be single issue, each node has a weight of one. The level of each node in the graph is also computed to be used in merge decisions. Each edge in the dependence graph is weighted according to the cost of cutting it. In its simplest form we assign weights to each edge as being the length of the longest path from a source to the sink node of which it is a part.

3.2.2 Partition Weights

The weight of a partition p is given by the number of nodes in it and the length of the critical path within this partition. Each partition also has a *flexibility* attribute. The flexibility of a partition is the number of instructions it can subsume without potentially increasing its sub-schedule length.

$$flex_p = \sum_{(i,j) \in cp_p} l(i,j) - |V_p| + 1 \tag{2}$$

The flexibilities of partitions are used heavily in the algorithm to decide whether candidate partitions should be merged or not.

3.3 Partitioning

3.3.1 Parallel Schedule Length

The object of partitioning is to minimize the overall schedule length. This length depends on the size of the largest partition as well as the levels at which the inter-partition edges occur. The levels of the crossing edges and flexibilities of partitions are the used to estimate the changes in schedule length if partitions are merged.

3.3.2 Algorithm

The pseudo code of the heuristic algorithm is given as Algorithm 1. The partitioning algorithm consists of four phases. The first phase initializes the data structures and annotates the dependence graph with the required weights. The second phase, which is similar to list scheduling maintains a working set of nodes and merges connected partitions only if necessary. In the third phase the algorithm reduces the number of partitions to be equal to the number of available processing cores. This happens on;y in the case where the number of partitions created in the second phase is greater than required. The fourth phase analyzes the benefits of parallelization and merges partitions if partitioning is not feasible.

The four phases of the algorithm are enumerated and described below. The algorithm takes the dependence graph of a code block and the number of processing cores available as parameters.

Algorithm 1 Partition ($DependenceGraph\ \mathcal{G}, Processors\ k$)

1: $\mathcal{P} \leftarrow \emptyset$
2: assignWeights($\mathcal{G}$)
3: $p_{cp} \leftarrow$ createPartition($cp_{\mathcal{G}}$)
4: $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_{cp}\}$
5: $\mathcal{A} \leftarrow$ available($\mathcal{G}$)
6: **while** $\mathcal{A} \neq \emptyset$ **do**
7: **for all** $a \in \mathcal{A}$ **do**
8: $p_a \leftarrow$ createPartition($\{a\}$)
9: $\mathcal{P} \leftarrow \mathcal{P} \cup \{p_a\}$
10: **end for**
11: mergePartitions($\mathcal{P}$) // conservative merge of connected partitions only
12: $\mathcal{A} \leftarrow$ available($\mathcal{G}$)
13: **end while**
14: **while** $|\mathcal{P}| > k$ **do**
15: mergeTwoPartitions($\mathcal{P}$) // aggressive merge of strongly connected partitions
16: **end while**
17: **repeat**
18: mergeNonParallelizablePartitions($\mathcal{P}$) // relaxed merge only iff no gains from parallelism
19: **until** no change in $\mathcal{P}$
20: **return** $\mathcal{P}$

- **Phase 1 (Lines 1-4) -** This is the initialization phase. The set of partitions is initialized to be empty. Weights are then assigned to the edges and nodes of the dependence graph. The function $assignweights(\mathcal{G})$ traverses the graph depth first and also sets the level of each node. Once the weight assignment is done, the algorithm uses the edge weights to find the critical path, creates a partition consisting of the nodes which are a part of the critical path and adds this partition to the list of partitions.

- **Phase 2 (Lines 5-13) -** This phase maintains a list $\mathcal{A}$ of nodes in the graph which are not yet assigned to a partition but all their ancestors are already part of some partition.

  In the loop that encompasses lines 6-13, a separate partition is created for each node in $\mathcal{A}$ and that partition is added to the partitions list $\mathcal{P}$. Before the completion of the loop iteration $mergePartitions()$ is called on the partitions list $\mathcal{P}$ and the work list $\mathcal{A}$ is updated.

  The merge in this phase (line 11), is extremely conservative and merges any two partitions which are *high connected*. Two partitions are high connected if there is an edge from the sink node of one partition to the source node of the other.

- **Phase 3 (Lines 14-16) -** This phase is only relevant in the case where the number of partitions generated by the last phase is greater than k, the number of available processing cores. If this is indeed the case then two partitions are chosen to be merged from the partition list which yield the greatest benefit in terms of parallelization and reduced cross-partition edges. This is likely to be the smallest partition with the largest number of outgoing edges to be merged with a connected partition with the greatest flexibility. At the end of this phase the total number of partitions is guaranteed to be less than or equal to k.

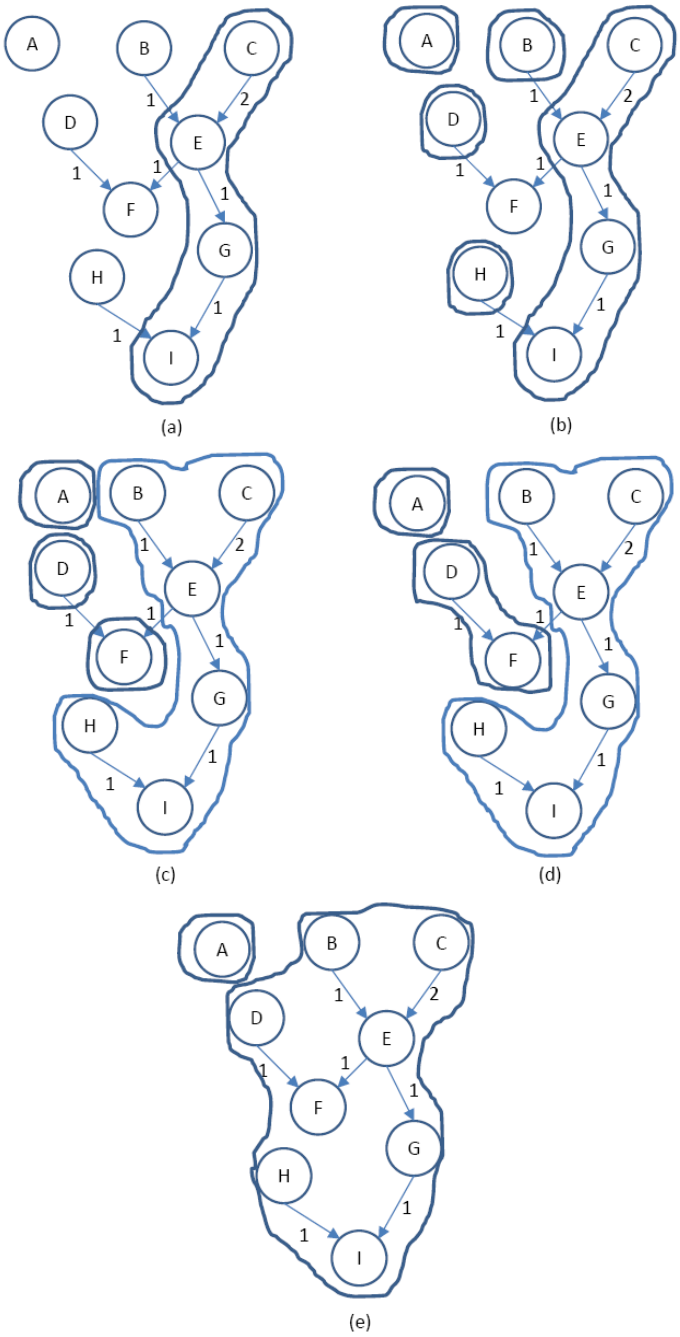

Figure 3: An example run of the Critical Path Heuristic algorithm on a basic block, with $k = 2$.

- **Phase 4 (Lines 17-19) -** In the last phase the algorithm attempts to merge non-parallelizable partitions. These are partitions for which executing the instructions in them on separate processing cores would have a longer schedule than if they were merged to form a single partition.

  The efficacy of partitioning in this phase is decided based on the connectedness of two partitions and their combined flexibility. This gives an estimate as to whether the merged partition would take fewer cycles to execute than the two separately. This phase is guaranteed to terminate as there has to be at least one partition in the list at the end.

Figure 3 illustrates the algorithm using an example basic block for $k = 2$. Partitions are shown as crooked lines going around nodes of the graph. Figure 3(a) shows the basic block graph after the first phase in which the critical path has been assigned to a partition. (b) and (c) illustrate the second phase which results in four partitions. (d) and (e) illustrate the third phase. The fourth phase does not result in any changes to the partitions as the two remaining partitions are disjoint in this particular case.

4 Experiments

The critical path heuristic algorithm was implemented using the Trimaran compiler framework [4].

4.1 Implementation and Experimental Setup

The Critical Path heuristic algorithm was implemented in the back-end of the Trimaran compiler framework, called Elcor. Trimaran is a publicly available C compiler and simulation infrastructure for supporting state of the art research in compilers. It is mainly used for instruction level parallelism (ILP) research for modern architectures. The system is oriented towards EPIC (Explicitly Parallel Instruction Computing) architectures modeled on VLIW, and supports compiler research in what is typically considered to be back-end techniques such as instruction scheduling, register allocation, and machine-dependent optimizations, as well as front end design using OpenIMPACT [17]. Figure 4 gives a high level picture of Trimaran and its components.

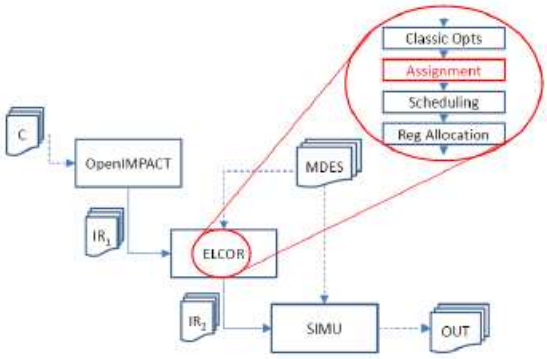

Figure 4: Implementation of Critical Path Heuristic partitioning in Trimaran.

Instruction assignment using the critical path heuristic has been implemented in the back end optimizer called Elcor. The partitioning phase is placed before the scheduling phase, and once partitioned the instructions are scheduled on the available cores using list scheduling.

For the purpose of the experiments two different machine configurations were used, one with two homogenous processing cores and the other having four. Operation latencies are similar to the Itanium and a perfect memory model is assumed. For each benchmark the number of compute cycles was used as the evaluation metric.

4.2 Evaluation and Results

To evaluate the performance of the heuristic algorithm (CPH), it was run on several benchmarks ranging from fibonacci computations to jpeg compression algorithms and public key encryption and authentication. The resultant speedups were compared against precious state-of-the-art greedy algorithm (BUG) and hierarchical partitioning algorithm (RHOP).

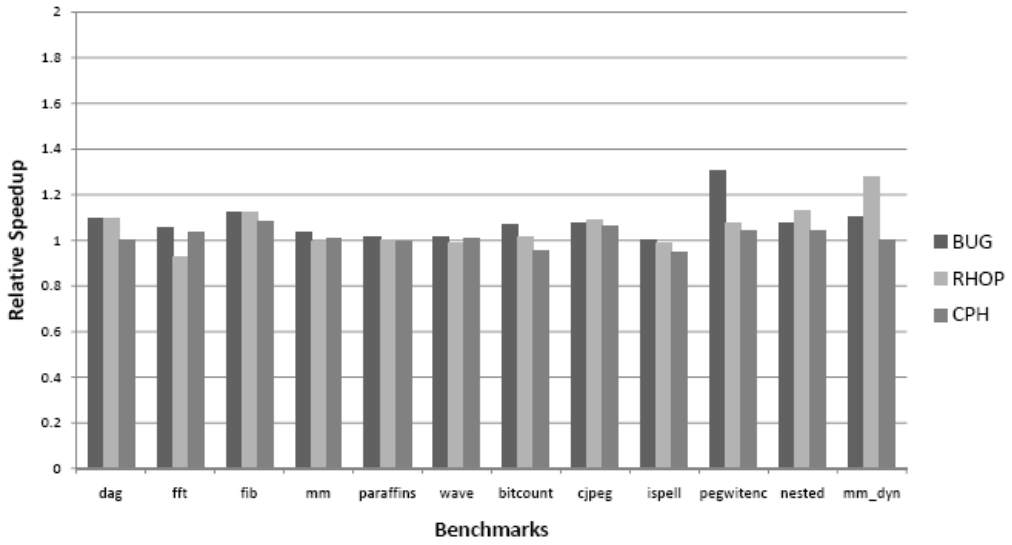

Figure 5: Basic blocks parallelized for two cores.

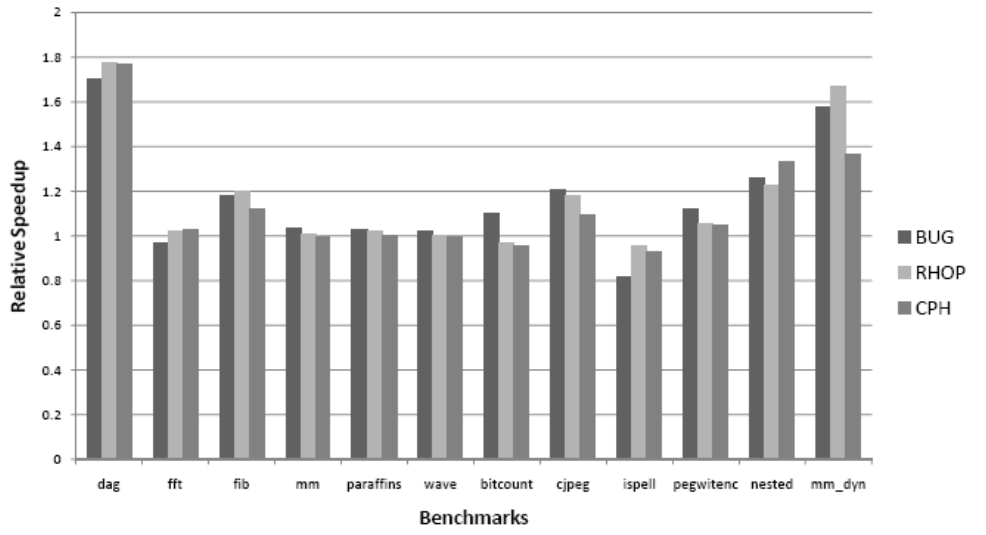

Figure 6: Hyper blocks parallelized for two cores.

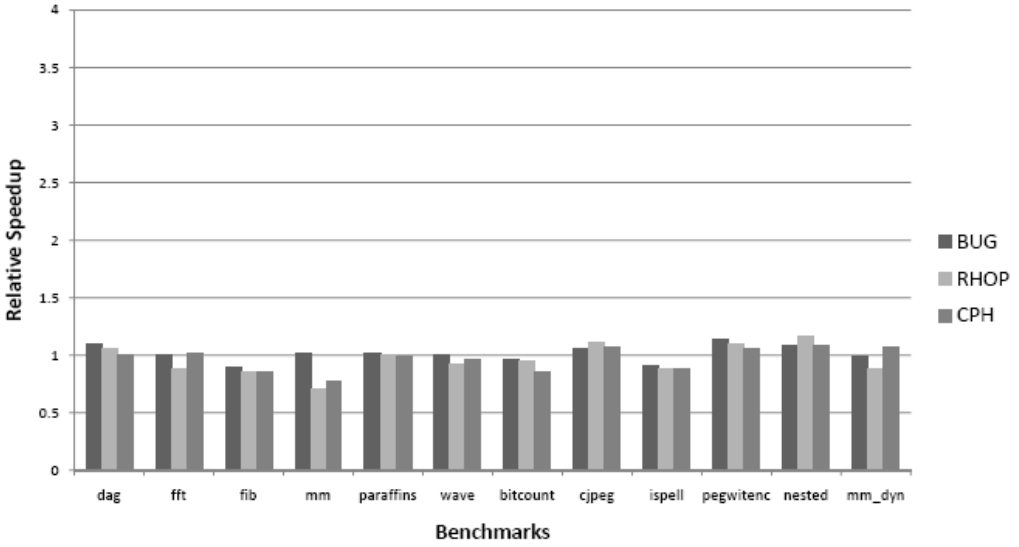

Figure 7: Basic blocks parallelized for four cores.

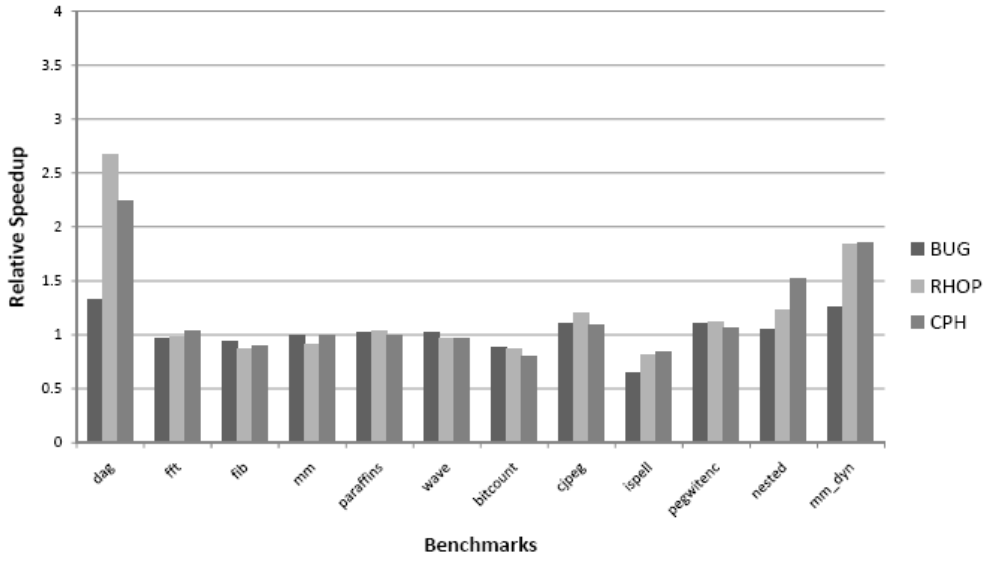

Figure 8: Hyper blocks parallelized for four cores.

Figure 5 shows the relative speedups for basic blocks on two cores. On average the speedups are less than 8%. CPH performs within a range of 5% of BUG and within 3% of RHOP speedups. Figure 6 shows the results of the same experiment conducted on hyper blocks. The average speedups in the case of hyper blocks are 15%, with CPH performing within 2% of both BUG and RHOP.

The same experiments were repeated for a four core configuration. The relative speedups are shown in Figures 7 and 8. In the case of basic blocks there is an average slowdown of 3%. For the basic block case CPH performs within 4% of BUG and is in fact 2% better than RHOP. In the case of hyper blocks the four core configuration results in speedups of around 20%. In this case CPH performs better than both BUG and RHOP. It shows improvements of 14% on average over the previous approaches on hyper blocks.

5 Related Work

Lee et al. [13] present a multi-heuristic framework for scheduling basic blocks, superb locks and traces. The technique is called *convergent scheduling*. The scheduler maintains a three dimensional weight matrix $W_{i,c,t}$, where the ith dimension represents the instructions, c spans over the number or processors and t spans over possible time slots. The scheduler iteratively executes multiple scheduling phases, each one of which, heuristically modifies the matrix to schedule each instruction on a processor for a specific time slot, according to a specific constraint. The main constraints are pre-placement, communication minimization and load balancing. After several passes the weights are expected to converge. The resultant matrix is used by a traditional scheduler to assign instructions to processors. The framework has been implemented on two different spatial architectures, Raw and clustered VLIW. The effectiveness of the framework was evaluated on standard benchmarks, mostly the ones with dense matrix code. An earlier attempt was made by the same group for scheduling basic blocks in the Raw compiler [12]. This technique iteratively clustered together instructions with little or no parallelism and then assigned these clusters to available processors. A similar approach was used to schedule instructions on a decoupled access/execute architectures [16]. These techniques seem to work well on selective benchmark suits with fine tuned system parameters which are configured using trial and error. It difficult to evaluate the actual effectiveness of these technique mainly because it attempts to solve the scheduling and assignment problems intermittently. In contrast our approach attempts to solve the assignment problem first.

The most well known solutions to the assignment problem are greedy and hierarchical partitioning algorithms which assign the instructions before the scheduling phase in the compiler. The bottom-up greedy, or BUG algorithm [9] proceeds by recursing depth first along the data dependence graph, assigning the critical paths first. It assigns each instruction to a processor based on estimates of when the instruction and its predecessors can complete execution at the earliest. These values are computed using the resource requirement information for each instruction. The algorithm queries this information before and after the assignment to effectively assign instructions to the available processors. This technique works well for simple graphs, but as the graphs become more complex the local nature of the greedy algorithm directs it to make decisions that negatively affect future decisions.

Chu et al. [6] describe a region-based hierarchical operation partitioning algorithm (RHOP), which is also a pre-scheduling method to partition operations on multiple processors. In order to produce a partition that can result in an efficient schedule, RHOP uses schedule estimates and a multilevel graph partitioner to generate cluster assignments. This approach partitions a data dependence graph based on weighted nodes and edges. The algorithm uses a heuristic to

assigns weights to the nodes to reflect their resource usage and to the edges to reflect the cost of inter-processor communication in case the two nodes connected by an edge are assigned to different processors. In the partitioning phase, nodes are grouped together by two processes called *coarsening* and *refinement* [10, 11]. Coarsening uses edge weights to group together operations by iteratively pairing them into larger groups while targeting the high weighted edges first. The coarsening phase ends when the number of groups is equal to the number of desired processors for the machine. The refinement phase improves the partition produced by the coarsening phase by moving nodes from one partition to another. The goal of this phase is to improve the balance between partitions while minimizing the overall communication cost. The moves are considered feasible if there is an improvement in the gain from added parallelism minus the cost of additional inter-processor communications. The algorithm has been implemented in the Trimaran framework. Subsequent work from the same group has attempted to partition data over multicore architectures with a more complex memory hierarchy [7, 8].

6 Discussion and Analysis

The evaluation of the given approach approach as well as previous approaches show that, in general there is a relative slowdown due to partitioning. The reason is that *cross-block edges* are not accounted for when local partitioning decisions are made on basic blocks and hyper blocks. Cross-block edges are ones that represent data dependencies crossing block boundaries. These are essentially ignored when partitioning decisions are made. Considering these edges during the partitioning phase amounts to a global partitioning technique which is significantly harder than the problem considered in this paper. On the other hand it would yield better results, even if the cross-block edges are considered within the heuristic to guide the partitioning algorithm.

The above observation also entails that a localized approach would also affect the optimal partitioning of basic blocks and hyper blocks. This means that even if we find the optimal assignments for the code blocks, the resultant speedups may be less than the heuristic ones.

On another note, further tuning of the critical path heuristic may improve its performance. There are several parameters which can be modified within the merge operations to increase the effectiveness of the algorithm.

7 Conclusions and Future Work

This paper describes a heuristic instruction assignment technique based on the assumption of non-parallelizability of the critical path. The algorithm partitions the directed acyclic data dependence graph representing a basic block or a hyper block of code. The algorithm is much simpler than its previous counterparts and much easier to implement. It uses schedule length estimates and flexibility within partitions in order to make merge decisions. The decisions are not localized but rather considers the region as a whole and converges to a solution by iteratively merging the most non-parallelizable partitions first.

The given approach has been evaluated and compared against two popular algorithms, the first being a greedy approach named BUG and the second being the hierarchical partitioning approach namely RHOP. Performance comparison reveals that on average the heuristic performs similar to previous algorithms for both two and four core machines. The key observation is that, in most

cases, speedups do not scale with increasing number of cores but the techniques work better with larger blocks compared to smaller ones.

As future work, the heuristic can be tuned further for better partitioning. In an orthogonal effort, optimal partitioning can be explored which is not restricted by the critical path.

References

[1] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. pages 483–485, Reston, VA, USA, 1967.

[2] Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. In *SPAA '04: Proceedings of the Sixteenth annual ACM symposium on Parallelism in Algorithms and Architectures*, pages 120–124, Barcelona, Spain, 2004.

[3] R. Berridge, R. M. Averill III, A. E. Barish, M. A. Bowen, P. J. Camporese, J. DiLullo, P. E. Dudley, J. Keinert, D. W. Lewis, R. D. Morel, T. Rosser, N. S. Schwartz, P. Shephard, H. H. Smith, D. Thomas, P. J. Restle, J. R. Ripley, S. L. Runyon, and P. M. Williams. IBM POWER6 Microprocessor Physical Design and Design Methodology. White paper, IBM, Inc., November 2007.

[4] Lakshmi N. Chakrapani, John C. Gyllenhaal, Wen mei W. Hwu, Scott A. Mahlke, Krishna V. Palem, and Rodric M. Rabbah. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. In *LCPC '04: Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004*, pages 32–41, 2004.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[6] Michael Chu, Kevin Fan, and Scott Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 300–311, New York, NY, USA, 2003. ACM.

[7] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 369–380, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Michael L. Chu and Scott A. Mahlke. Compiler-directed Data Partitioning for Multicluster Processors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 208–220, Washington, DC, USA, 2006. IEEE Computer Society.

[9] John R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press, Cambridge, MA, USA, 1986.

[10] Bruce Hendrickson and Robert Leland. A Multilevel Algorithm for Partitioning Graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 28, New York, NY, USA, 1995. ACM.

[11] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[12] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 46–57, New York, NY, USA, 1998. ACM.

[13] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent Scheduling. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 111–122, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[14] Matther Monteyne. RapidMind Multi-Core Development Platform. White paper, RapidMind, Inc., February 2008.

[15] R M Ramanathan. Intel Multi-Core Processors. White paper, Intel Corporation, Inc., August 2006.

[16] Kevin D. Rich and Matthew K. Farrens. Code Partitioning in Decoupled Compilers. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1008–1017, London, UK, 2000. Springer-Verlag.

[17] John W. Sias, Sain zee Ueng, Geoff A. Kent, Ian M. Steiner, Erik M. Nystrom, and Wen mei W. Hwu. Field-testing IMPACT EPIC Research Results in Itanium 2. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 26, Washington, DC, USA, 2004. IEEE Computer Society.