Global Multi-Threaded Instruction Scheduling

Guilherme Ottoni David I. August

Department of Computer Science
Princeton University
{ottoni, august}@princeton.edu

Abstract

Recently, the microprocessor industry has moved toward chip multiprocessor (CMP) designs as a means of utilizing the increasing transistor counts in the face of physical and micro-architectural limitations. Despite this move, CMPs do not directly improve the performance of single-threaded codes, a characteristic of most applications. In order to support parallelization of general-purpose applications, computer architects have proposed CMPs with light-weight scalar communication mechanisms [21, 23, 29]. Despite such support, most existing compiler multi-threading techniques have generally demonstrated little effectiveness in extracting parallelism from non-scientific applications [14, 15, 22]. The main reason for this is that such techniques are mostly restricted to extracting parallelism within straight-line regions of code.

In this paper, we first propose a framework that enables global multi-threaded instruction scheduling *in general. We then describe GREMIO, a scheduler built using this framework. GREMIO operates at a* global scope*, at the procedure level, and uses control dependence analysis to extract* non-speculative *thread-level parallelism from sequential codes. Using a fully automatic compiler implementation of GREMIO and a validated processor model, this paper demonstrates gains for a dual-core CMP model running a variety of codes. Our experiments demonstrate the advantage of exploiting global scheduling for multi-threaded architectures, and present gains in a detailed comparison with the Decoupled Software Pipelining (DSWP) multi-threading technique [18]. Furthermore, our experiments show that adding GREMIO to a compiler with DSWP improves the average speedup from 16.5% to 32.8% for important benchmark functions when utilizing two cores, indicating the importance of this technique in making compilers extract threads effectively.*

1 Introduction

In the last few years, the microprocessor industry has been undergoing one of its most fundamental changes in decades. Suddenly, hard physical limitations, aligned with the diminishing returns of micro-architectural improvements, have prevented the design of faster microprocessors. Nevertheless, the number of transistors available on a chip continues to increase exponentially over time. Combined, these factors have directed all major microprocessor manufacturers toward multi-core designs, also known as chip multiprocessors (CMPs). Unfortunately, while CMPs increase throughput for multiprogrammed and multi-threaded codes, many important applications are single-threaded and thus do not benefit from CMPs.

This change in paradigm has resulted in a tremendous interest in parallel applications. Although ideally programmers could rewrite all applications in a parallel paradigm, parallel programming has long been recognized as more time-consuming, error-prone, and harder to debug than its sequential counterpart. Furthermore, it is impractical to rewrite all the existing applications. A less costly alternative would be to use parallelizing compilers to automatically generate parallel code from sequential programs. Unfortunately, despite decades of research on parallelizing compilers, these have only proved effective in the restricted domain of scientific applications, which often have regular array-based memory accesses and little control flow.

Computer architects have worked in at least three directions in order to support thread-level parallelism (TLP) extraction for general-purpose applications. First, various hardware mechanisms to support transactional memory have been studied. Although transactions provide a nice abstraction to express parallelism, the programmer is still required to express the parallel sections of code. Second, several mechanisms to exploit thread-level speculation (TLS) have been proposed [27, 28]. Though effective in many cases, these techniques generally require expensive hardware support. Furthermore, these techniques are complementary to non-speculative compiler techniques. For example, the dependence analysis used in this paper can be employed to guide thread spawning for TLS, or to speed up individual iterations in a loop to which TLS is applied. Third, computer architects have proposed hardware mechanisms to lower the inter-thread communication costs, thus enabling the exploitation of fine-grained TLP found in general-purpose applications [20, 21, 23, 29]. These mechanisms typically consist of an on-chip interconnect between the processor cores and means to communicate scalar values

from one core to another. To the software, these communication mechanisms look like sets of queues with blocking primitives to send and receive values, typically in the form of special `produce` and `consume` instructions or register-mapped queues. Extracting parallelism for these processors consists of partitioning the computation into threads and generating code to use the hardware communication support to satisfy inter-thread dependences. The parallelism exposed by these processors is of finer granularity than what is typically exploited by programmers of parallel systems, making it even harder to manually exploit these opportunities. Therefore, generating code that exploits this parallelism is better performed by a compiler's instruction scheduler.

Instruction scheduling techniques, akin to other compiler optimizations, can be classified according to the scope of the regions they operate at a time. For the purpose of this paper, we use the following classification: *local* techniques, operating at basic blocks or traces; *loop* techniques, operating at loops; and *global* techniques, operating at whole procedures. Many existing multi-threaded scheduling techniques are mostly based on local scheduling [14, 15, 22]. We call these techniques *local multi-threaded* (LMT) scheduling. In these techniques, all threads synchronously execute every basic block or trace, thus using threads to simply exploit *instruction-level* parallelism within straight-line regions of code.

One of our key observations is that LMT scheduling techniques do not take advantage of a main feature of multi-threaded architectures: the ability to simultaneously follow different execution paths in different processor cores. In fact, several limit studies have shown that exploiting parallelism beyond local regions of code and executing multiple flows of control in parallel are necessary to extract reasonable amounts of parallelism from most applications [13]. As an example, consider the sample C code in Figure 1. Although these loops may run for a large number of iterations, very little instruction-level parallelism is available within each basic block. For such control-intensive codes, LMT scheduling techniques cannot extract any parallelism. Notice, however, that the computation in each loop is independent, and therefore they can be executed in parallel. Nevertheless, in order to exploit such sources of parallelism, it is necessary to perform *global multi-threaded* (GMT) scheduling.

The Decoupled Software Pipelining (DSWP) technique [18] can be viewed as a restricted form of GMT instruction scheduling, which is only applicable to loop nests in which pipeline parallelism is available. As shown in our experiments, DSWP misses opportunities because it only looks for pipeline multi-threading (PMT) and does not allow other forms of parallelism. GREMIO, the technique proposed in this paper, generalizes DSWP in the sense that

```
s1 = 0;
s2 = 0;
for (p = head; p != NULL; p = p->next) {
  s1 += p->value;
}
for (i = 0; a[i] != 0; i++){
  s2 += a[i];
}
printf("%d\n", s1 * s1 / s2);
```

Figure 1. Example code in C.

it is applicable to arbitrary code regions and that it can extract other forms of multi-threading (MT) parallelism. Nevertheless, our experiments demonstrate that each of DSWP and GREMIO is able to extract parallelism that the other is not. Therefore, they can play complementary roles in a multi-threading compiler.

This paper:

1. Introduces the general concept of GMT instruction scheduling.
2. Presents the algorithms used by GREMIO, a GMT instruction scheduler. These include a novel GMT list scheduling heuristic, and an effective dynamic programming algorithm to efficiently handle large code regions composed of complex loop nests.
3. Discusses a general approach to generate multi-threaded code from arbitrary partitions of the instructions among the threads, for any code region. This MT code generation technique is used by GREMIO and generalizes the one used for loop scheduling in [18] to operate on arbitrary CFGs.
4. Shows promising initial experimental results targeting a highly accurate dual-core Itanium 2 model.

The rest of the paper is organized as follows. Section 2 gives some background on Program Dependence Graphs (PDGs), a key program representation used in this work, and then discusses how PDGs are useful to enable generalized MT instruction scheduling. The scheduling algorithms used by GREMIO are presented in Section 3. In Section 4, we present experimental results. Finally, we discuss related work in Section 5, and conclude in Section 6.

2 Program Dependences and Multi-Threaded Instruction Scheduling

Program dependences constitute an important abstraction in compiler optimization and parallelization. This section first provides some background on Program Dependence Graphs (PDG), and then demonstrates how PDGs enable global MT instruction scheduling.

2.1 Program Dependence Graphs

Local scheduling techniques operate by constructing a *data dependence graph* representing all data dependences that must be respected. At a low-level representation, data dependences can take two forms: register data dependences, or memory data dependences. Furthermore, data dependences can be of three kinds, depending on whether the involved instructions read or write the data location: *flow dependence*, which goes from a write to a read; *anti-dependence*, which goes from a read to a write; and *output dependence*, which goes from a write to another write. Register data dependences can be efficiently and precisely computed through data-flow analysis. For memory data dependences, compilers typically rely on the result of pointer analysis to determine which loads and stores may access the same memory locations. Although computationally much more complicated, practical existing pointer analysis can typically disambiguate a large number of non-conflicting memory accesses even for type-unsafe languages like C (e.g. [2]).

The key addition from a local to a global scheduling technique is the necessity of handling control flow. In other words, in addition to the data dependences typically used for local scheduling, it is necessary to add *control dependence* arcs to the dependence graph. Although slightly different definitions of control dependence exist, the most general one, stated in Definition 1 below, was introduced by Ferrante et al. [6]. This definition is meaningful even at lower-level program representations (typically used for instruction schedulers), since it is not based on the syntactical structure of the program. Instead of syntactical constructors, it uses the *post-dominance* relation [16].

Definition 1 (Control Dependence). *Let G be a CFG, and X and Y two nodes in G. Y is control dependent on X iff:*

1. *there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y; and*
2. *X is not strictly post-dominated by Y.*

Cytron et al. [4] proposed an efficient algorithm to compute control dependences according to this definition.

Dependence graphs including both data and control dependences are generally called *Program Dependence Graphs* (PDGs). PDGs are widely used in compilers as an intermediate representation due to several important properties. In particular, Horwitz et al. [10] proved, using syntax-based control dependences, the *Equivalence Theorem*. According to this theorem, two programs with the same PDG are equivalent. Later, Sarkar [25] proved a similar result for PDGs using control dependences according to Definition 1.

As an example, consider the low-level representation in Figure 2(a) of Figure 1. Figures 2(b)-(d) illustrate the corresponding CFG, post-dominance tree, and PDG. In Figure 2(d), solid arcs represent register data dependences, and the dotted arcs represent control dependences in the PDG.

2.2 PDGs and Multi-Threaded Instruction Scheduling

Following Sarkar's [25] result, every instruction scheduling technique needs to preserve all dependences in a PDG. For multi-threaded scheduling, this implies that instructions to synchronize and communicate values among the threads have to be inserted in the code in order to satisfy inter-thread dependences.

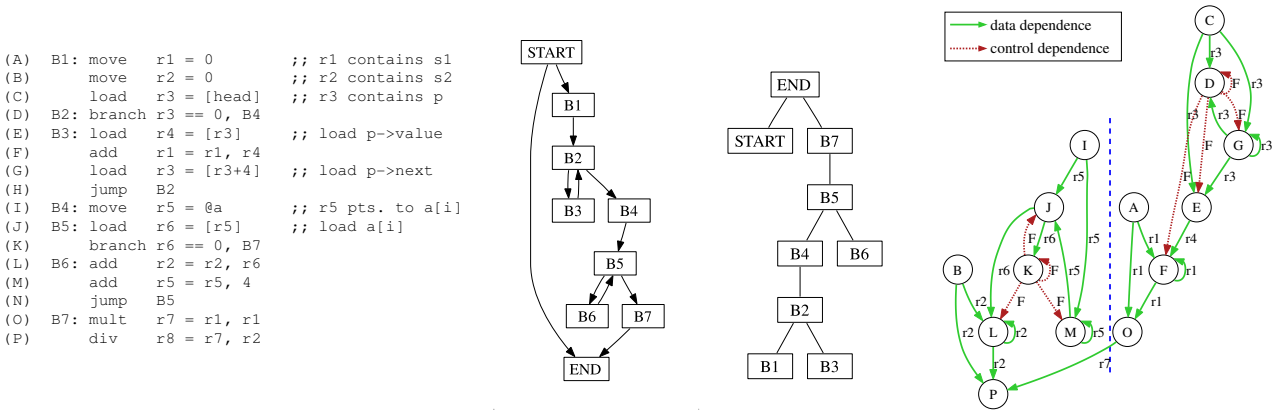
For LMT instruction scheduling techniques, the preservation of all program's dependences is guaranteed in two ways. First, the schedule of instructions inside basic blocks or traces makes sure that data dependences are satisfied. This requires explicit communication instructions to be inserted if the source and sink of a dependence are in different threads. Second, all threads execute every basic block or trace in synchrony. This is achieved by having the direction of each branch communicated from the thread containing it to all other threads [14, 15, 22].

The key to enabling *global* MT instruction scheduling is in allowing the threads to concurrently execute different regions of code, while guaranteeing that all data and control dependences are respected. The GMT scheduler should be able to deal with *arbitrary code regions*, and should be able to make *any partition decisions* while still generating correct code.

Ottoni et al. [18] describe a MT instruction scheduling technique to extract pipeline parallelism from *loop* regions. Their technique chooses a partition of the code that will form a pipeline of threads, and then applies a novel MT code generation algorithm. Briefly, this code generation algorithm has four main steps. For each of the threads specified by the partition, a new CFG is generated with only the necessary basic blocks for this thread. Then, the instructions are inserted in the CFG for the thread to which they were assigned. Next, the necessary inter-thread communication and synchronization instructions are inserted into the code. Finally, branch and jump instructions are adjusted to account for missing basic blocks in the new CFGs.

Even though the scheduling technique of [18] is limited to loops, its MT code generation algorithm is more general. In fact, it can be applied to any region with a *reducible* CFG. Furthermore, what prevents their technique from being applicable to irreducible CFGs is the *dependence graph* they use, and not the code generation algorithm itself.

The *Dependence Graph* (DG) used in [18] contains both data and control dependences, much like a PDG. For control dependences, however, the DG also includes *loop-iteration control dependences* (Section 2.3.1 in [18]). The authors argue that, because communication queues are reused every loop iteration, these dependences are neces-



(a) Low-level IR (b) CFG (c) post-dom. tree (d) PDG

Figure 2. For the example in Figure 1: (a) low-level IR, (b) CFG, (c) post-dominance tree, and (d) PDG.

sary to avoid mixing values from two different loop iterations. The limitation in that paper is that loop-iteration control dependences are defined in terms of loops in the program. For this reason, in an irreducible CFG, cyclic regions of code that are not loops would cause a problem.

However, it turns out that these loop-iteration control dependences are only necessary because of the notion of control dependences used in [18], which is the one used in the IMPACT compiler [26]. IMPACT's control dependences do not contain *loop-carried* control dependences, which naturally exist using Definition 1. Ferrante et al.'s control dependence definition not only gives the smallest set of control dependences that need to be respected in a program [25], but is also defined for arbitrary CFGs. Therefore, substituting the control dependences in Ottoni et al.'s DG by Ferrante et al.'s control dependences enables the code generation algorithm of [18] to be used on arbitrary regions of code.

Theorem 1. *The Multi-Threaded Code Generation in [18] preserves all the dependences in a PDG.*

Proof. In interest of space, we just provide a sketch of this proof. Intra-thread data dependences are trivially satisfied because the instructions are inserted into the new CFGs in the same relative order as in the original code. The trickiest part is proving that control dependences are preserved after adjusting the branch targets to the closest relevant post-dominator (step 4 in Section 2.2.3 of [18]). This can be demonstrated by showing that the post-dominance relation is preserved among corresponding blocks in the new CFGs. From that, inter-thread data dependences are satisfied because the condition of execution of each instruction is preserved, and the data communication is inserted right after the source of the dependence. □

From Theorem 1, the correctness of the multi-threaded code generation algorithm then follows immediately:

Theorem 2. *The Multi-Threaded Code Generation in [18] preserves the semantics of the original code.*

Proof. From Theorem 1, all PDG dependences are preserved. Therefore, using the Equivalence Theorem [25], the semantics of the original program is preserved. □

According to Theorem 2, combining Ottoni et al.'s MT code generation with Ferrante et al.'s PDG provides a framework to perform GMT instruction scheduling in general. With it, *any* partitioning heuristic can be applied to the PDG, and the MT code generation algorithm will produce correct code. In this framework, the DSWP technique [18] can effectively be regarded as a partitioning algorithm that only extracts pipeline MT parallelism from loops. In the following section, we present a more general MT partitioning technique called GREMIO, which is not limited to loops and that can extract other kinds of MT parallelism. For example, for the PDG in Figure 2(d), GREMIO partitions the code into two threads as depicted by the vertical dashed line. This partition corresponds to scheduling each loop of Figure 1 into a separate thread.

3 Global Multi-Threaded Instruction Scheduling Algorithms

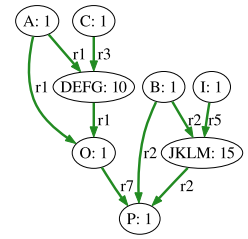
This section describes GREMIO's algorithms in detail, and illustrates them on the example of Figure 1. Although a MT instruction scheduler can be combined with a traditional single-threaded scheduler, we opted not to do so in this work. One reason for this is that the MT code generated by GREMIO can be further optimized before the actual assembly code generation. Additionally, exposing all the

machine details to GREMIO would make its implementation more complex. Instead, we preferred to keep GREMIO simpler by providing it with just a few key characteristics of the target processor, namely the number of threads and the issue-width of the processor. A latency of one cycle is assumed for most instructions (except for function calls), and no information about structural hazards is used.

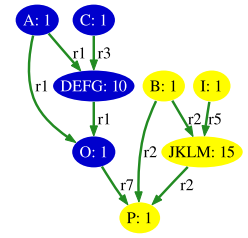
GREMIO uses the PDG as an intermediate representation not only for MT code generation, but also for scheduling decisions. Using the PDG to guide scheduling decisions is attractive because it makes explicit the communications that will be incurred. In other words, scheduling two dependent instructions to different threads will require an inter-thread communication. A problem that arises from using a PDG for scheduling decisions is the presence of cycles. The PDG for an arbitrary code region can have cycles due to loops in the CFG and loop-carried dependences. Scheduling cyclic graphs is more complicated than scheduling acyclic graphs. This is because the goal of a scheduler is to minimize the critical (i.e. longest) path through the graph. Although scheduling of acyclic graphs in the presence of resource constraints is NP-hard, at least finding the critical path in such graphs can be solved in linear time, through a topological sort. For cyclic graphs, however, even finding the longest path is NP-hard [8].

Given the inherent difficulty of the global scheduling problem for cyclic code regions, GREMIO uses a simplifying approach that reduces it to the acyclic scheduling problem, for which well-known heuristics based on list scheduling [16] exist. In order to reduce the cyclic scheduling problem to an acyclic one, GREMIO uses two simplifications to the problem. First, when scheduling a given code region, each of its inner loops is coalesced to a single node, with an aggregated latency that assumes its average number of iterations (based on profiling or static estimates). Secondly, if the code region being scheduled is itself a loop, all its loop-carried dependences are disregarded. To deal with the possibility of irreducible code, a loop hierarchy that includes irreducible loops is used [9]. It is important to note that these simplifying assumptions are used for partitioning decisions only; the MT code generation algorithm takes *all* dependences into account to generate correct code.

To distinguish from a full PDG, we call the dependence graph for a region with its inner loops coalesced and its loop-carried dependences ignored a *Hierarchical Program Dependence Graph* (HPDG). In a HPDG, the nodes represent either a single instruction, called a *simple node*, or a coalesced inner loop, called a *loop node*. Figure 3(a) illustrates the HPDG corresponding to the PDG from Figure 2(d). The nodes are labeled by their corresponding nodes in the PDG, followed by their estimated execution latency. There are only two loop nodes in this example: DEFG and JKLM.



(a) HPDG



(b) Clustered HPDG

	Core 0		Core 1	
cycle	issue 0	issue 1	issue 0	issue 1
0	A	C	B	I
1	D E	F G	J K	L M
2	D E	F G	J K	L M
3	D E	F G	J K	L M
4	D E	F G	J K	L M
5	D E	F G	J K	L M
6	D E	F G	J K	L M
7	D E	F G	J K	L M
8	D E	F G	J K	L M
9	D E	F G	J K	L M
10	D E	F G	J K	L M
11	O		J K	L M
12	prod r7		J K	L M
13			J K	L M
14			J K	L M
15			J K	L M
16			cons r7	
17			P	

(c) Virtual Schedule

Figure 3. Operation of GREMIO on the example from Figure 2.

Another complication intrinsic to MT scheduling is that the generated threads need to communicate to satisfy dependences, and so it is necessary to take the communication overhead into account while making scheduling decisions[1]. For instance, even though two instructions can be executed in parallel on different threads, this might not be profitable due to the overhead to communicate their operands. To address this problem, GREMIO uses a *clustering* pre-scheduling pass on the HPDG, which takes into account the inter-thread communication overhead. The goal of this pass is to cluster together HPDG nodes that are likely to not benefit from schedules that assign them to different threads. Section 3.1 explains the clustering algorithm used by GREMIO, and Section 3.2 describes its partitioning heuristic.

3.1 Clustering Algorithm

There exist a variety of clustering algorithms in the parallel computing literature. These algorithms are used for task scheduling, being applicable to arbitrary directed acyclic graphs (DAGs). Therefore, because we reduced the original cyclic scheduling problem (on a PDG) to an acyclic problem (on a HPDG), we can rely on previous research on DAG-clustering algorithms.

We chose to use the *Dominant Sequence Clustering (DSC)* algorithm [32], which has been shown to be very effective and efficient. Efficiency is important here because,

[1] This complication also arises in single-threaded instruction scheduling for architectures with partitioned register files.

given the fine granularity of the nodes in a HPDG, their number can be very large (on the order of thousands).

DSC, like other clustering algorithms, groups nodes in clusters so that nodes in the same cluster are unlikely to benefit from executing in parallel. Therefore, all nodes in the same cluster should be scheduled on the same processor (thread here). DSC also assumes that each cluster will be executed on a different processor. Later, the scheduling pass can assign multiple clusters on the same thread to cope with a smaller number of processors.

Briefly, DSC operates as follows. In the beginning, each node is assigned to its own cluster. The critical path passing through each node of the graph is then computed, considering both the execution latencies of nodes and the communication latencies. The communication latency is assumed to be zero if and only if the nodes are in the same cluster. DSC then processes each node at a time, following a topological order prioritized by the nodes' critical path length. At each step, the benefit of merging the node being processed with each of its predecessors is analyzed. The advantage of merging a node with another cluster is that the communication latency from nodes in that cluster will be saved. The downside of merging is that the nodes assigned to the same cluster are assumed to execute sequentially, in the order they are added to the cluster. Therefore, the delayed execution after a merge may outweigh the benefits of the saved communication. The node being processed is then merged with its predecessors' cluster that reduces this node's critical path the most. If all such merges increase the critical path, this node is left alone in its own cluster. For our running example, Figure 3(b) illustrates the clusters resulting from DSC assuming a 2-cycle communication latency.

3.2 Global Multi-Threaded List Scheduling

After the clustering pass on the HPDG, the actual scheduling decisions are made. Here again, because of the reduction to an acyclic scheduling problem, we can rely on well-known acyclic scheduling algorithms. In particular, GREMIO uses a form of *list scheduling* with resource constraints, with some adaptations to better deal with our problem. This section describes list scheduling and the enhancements used by GREMIO.

The basic list scheduling algorithm assigns priorities to nodes and schedules each node following a prioritized topological order. Typically, the priority of a node is computed as the longest path from it to a leaf node. A node is scheduled at the earliest time that satisfies its input dependences and that conforms to the currently available resources.

GREMIO uses a variation of list scheduling to partition the HPDG into threads. Even though the HPDG is acyclic, control flow still poses additional complications to GMT list scheduling that do not exist in local list scheduling. When scheduling a basic block, local schedulers have the guarantee that all instructions will either execute or not. In other words, all instructions being scheduled are *control equivalent*. Therefore, as long as the dependences are satisfied and resources are available, the instructions can safely be issued simultaneously. The presence of arbitrary control flow complicates the matters for GMT scheduling. First, control flow causes many dependences not to occur during the execution. Second, not all instructions being scheduled are control equivalent. For example, the fact that an instruction X executes may not be related to the execution of another instruction Y, or may even imply that Y will not execute. To deal with the different possibilities, we introduce three different *control relations* among instructions, which are used in GREMIO's list scheduling.

Definition 2 (Control Relations). *Given two HPDG nodes X and Y, we call them:*

1. Control Equivalent*, if both X and Y are simple nodes with the same input control dependences.*
2. Mutually Control Exclusive*, if the execution of X implies that Y does not execute, or vice-versa.*
3. Control Conflicting*, otherwise.*

To illustrate these relations, consider the HPDG from Figure 3(a). In this example, A, B, C, I, O, and P are all control equivalent. Nodes DEFG and JKLM are control conflicting with every other node. No pair of nodes is mutually control exclusive in this example.

Although GREMIO uses list scheduling simply to decide the partition and applies the MT code generation later, it still builds a schedule of HPDG nodes to cycles. This schedule is not realistic in that it includes all the nodes in a HPDG, even though some of them are mutually control exclusive. For this reason, we call it a *virtual schedule*, and we say the nodes are scheduled on *virtual cycles*.

For traditional, single-threaded instruction scheduling, the resources correspond to the processor's functional units. To simplify the discussion, although GREMIO can be applied in general, we assume a CMP with each core single-threaded. In this scenario, there are two levels of resources: the target processor contains multiple cores, and each core has a set of functional units. Considering these two levels of resources, instead of simply assuming the total number of functional units in all cores, is important for many reasons. First, it enables us to consider the communication overhead to satisfy dependences between instructions scheduled on different cores. Furthermore, it allows us to benefit from key opportunities available in *multi-threaded* scheduling: the simultaneous issue of control-conflicting instructions. Because each core has its own control unit, control-conflicting instructions can be issued in different cores in the same cycle.

Thread-level scheduling decisions are made when scheduling the first node in a cluster. At this point, the best thread is chosen for that particular cluster, given what has already been scheduled. When scheduling the remaining nodes of a cluster, GREMIO simply schedules them on the thread previously chosen for this cluster.

The choice of the best thread to schedule a particular cluster to takes into account a number of factors. Broadly speaking, these factors try to find a good balance between two conflicting goals: maximizing the parallelism, and minimizing the inter-thread communication. For each thread, the total overhead of assigning the current cluster to it is computed. This total overhead is the sum of the following components:

1. *Communication Overhead*: this is the total number of cycles that will be necessary to satisfy dependences between this cluster and instructions in clusters already scheduled on different threads. This accounts for both overhead inside the cores (extra `produce` and `consume` instructions) and communication delay.
2. *Conflict Overhead*: this is the estimated number of cycles by which the execution of this cluster will be delayed when executing in this thread, considering the current load of unfinished instructions in clusters already assigned to this thread. This considers the both resource conflicts in terms of functional units, as well as control conflicts among instructions.

Once GREMIO chooses the thread to schedule a HPDG node to, it is necessary to estimate the virtual cycle in which that node can be issued in this core. The purpose of assigning nodes to virtual cycles within a thread is to guide the scheduling of the remaining nodes.

In order to find the virtual cycle in which a node can be issued in the chosen thread, it is necessary to consider two restrictions. First, it is necessary to make sure that the node's input dependences will be satisfied at the chosen cycle. For inter-thread dependences, it is necessary to account for the communication latency and corresponding `consume` instructions overhead. Second, the chosen cycle must be such that there are available resources in the chosen core, given the other nodes already scheduled on it. However, not all the nodes already scheduled on this thread should be considered. Resources used by nodes that are mutually control exclusive to this one are considered available since these nodes will never be issued simultaneously. On the other hand, the resource utilization of control equivalent nodes must be taken into account. Finally, the node cannot be issued in the same cycle as any previously scheduled node that has a control conflict with it. This is because each core has a single control unit, but control-conflicting nodes have unrelated conditions of execution. Notice that for target cores supporting predicated execution, however, this is not necessarily valid: two instructions with different execution conditions may be issued in parallel. But even for cores with predication support, loop nodes cannot be issued with anything else.

We now show how GREMIO's list scheduling algorithm works on our running example. For illustration purposes, we use as target a dual-core processor that can issue two instructions per cycle in each core (see Figure 3(c)). The list scheduling algorithm processes the nodes in the clustered HPDG (Figure 3(b)) in topological order. The nodes with highest priority (i.e. longest path to a leaf) are B and I. B is scheduled first, and it is arbitrarily assigned to core 1's first slot. Next, node I is considered and, because it belongs to the same cluster as B, the core of choice is 1. Because there is an available resource (issue slot) in core 1 at cycle 0, and the fact that B and I are control equivalent, I is scheduled on core 1's issue slot 1. At this point, either nodes A, C, or JKLM may be scheduled. Even though JKLM has the highest priority, its input dependences are not satisfied in the cycle being scheduled, cycle 0. Therefore, JKLM is not a *candidate* node in the current cycle. So node A is scheduled next, and the overheads described above are computed for scheduling A in each thread. Even though thread 1 (at core 1) has lower communication overhead (zero), it has higher conflict overheads. Therefore, core 0 is chosen for node A. The algorithm then proceeds, and the remaining scheduling decisions are all cluster-based. Figure 3(c) illustrates the final schedule built and the partitioning of the instructions among the threads.

3.3 Handling Loop Nests

Although GREMIO's scheduling algorithm follows the clusters formed a priori, an exception is made when handling inner loops. The motivation to do so is that inner loops may fall on the region's critical path, and they may also benefit from execution on multiple threads.

GREMIO handles inner loops as follows. For now, assume that it has an estimate for the latency to execute one invocation of an inner loop L_j using a number of threads i from 1 up to the number N of threads on the target processor. Let $latency_{L_j}(i)$, $1 \leq i \leq N$, denote these latencies. Considering L_j's control conflicts, the algorithm computes the cycle in which each thread will finish executing L_j's control-conflicting nodes already scheduled on it. From that, the earliest cycle in which a given number of threads i will be available for L_j can be computed, being denoted by $cycle\_available_{L_j}(i)$, $1 \leq i \leq N$. With that, the algorithm chooses the number of threads k on which to schedule this loop node such that $cycle\_available_{L_j}(k) + latency_{L_j}(k)$ is minimized. Intuitively, this will find the best balance between the wait to have more threads available and the benefit from executing the loop node on more threads. If more than k threads are available at $cycle\_available_{L_j}(k)$ (i.e., in

case multiple threads become available at this cycle), then the algorithm picks the k threads among them with which the loop node has more affinity. The affinity is computed as the number of dependences between this loop node and nodes already scheduled on each thread.

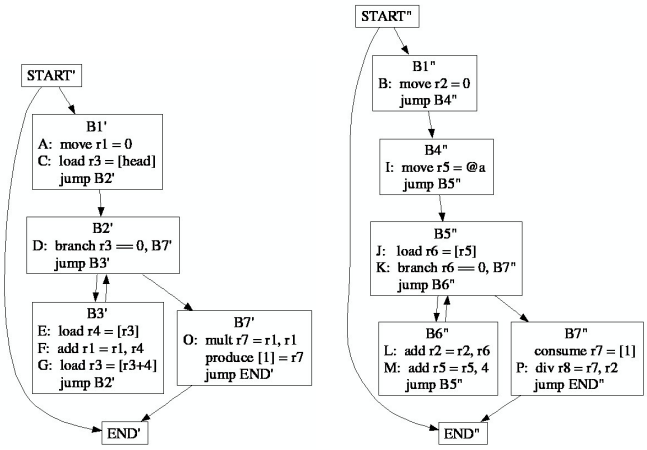
The question that remains now is: how are the $latency_{L_j}(i)$ values for each child loop L_j in the HPDG computed? Intuitively, this is a recursive question, since the same algorithm can be applied to the child loop L_j, targeting i threads, in order to compute $latency_{L_j}(i)$. This naturally leads to a recursive solution. Even better, dynamic programming can efficiently solve this problem in polynomial time. However, since this constrained scheduling problem is NP-hard, this dynamic programming approach may not be optimal because the list scheduling algorithm applied to each node is not guaranteed to be optimal.

More specifically, GREMIO's dynamic programming solution works as follows. First, it computes the loop hierarchy for the region to be scheduled. This can be viewed as a loop tree, where the root represents the whole region (which need not be a loop). The algorithm then proceeds bottom-up on this loop tree and, for each tree node L_j (either a loop or the whole region), it applies the GMT list scheduling algorithm to compute the latency to execute one iteration of that loop, with a number of threads i varying from 1 to N. The latency returned by the list scheduling algorithm is then multiplied by the average number of iterations per invocation of this loop, resulting in the $latency_{L_j}(i)$ values to be used for this loop node when scheduling its parent. In the end, the algorithm chooses the best schedule for the whole region by picking the number of threads k for the loop tree's root, R, such that $latency_R(k)$ is minimized. The corresponding partitioning of instructions onto threads can be obtained by keeping and propagating the partition $partition_{L_j}(i)$ of instructions corresponding to the value of $latency_{L_j}(i)$.

We note that this dynamic programming approach can be used in a general framework that considers other loop parallelization and scheduling techniques, such as DOALL, DOACROSS, and DSWP [18], besides the GMT list scheduling described here. However, the evaluation of such a general framework is beyond the scope of this paper.

3.4 Putting It All Together

After the partition into threads is chosen, the MT code generator algorithm discussed in Section 2.2 is applied. Figures 4(a)-(b) illustrate the generated code for the two threads corresponding to the global schedule depicted in Figure 3(c). As can be verified, each of the resulting threads contains only its relevant basic blocks, the instructions scheduled to it, the instructions inserted to satisfy the inter-thread dependences, and jumps inserted to connect the CFG. In this example, there is a single pair of



(a) Code for thread 1. (b) Code for thread 2.

Figure 4. Resulting multi-threaded code.

`produce` and `consume` instructions, corresponding to the only cross-thread dependence in Figure 2(d).

By analyzing the resulting code in Figures 4(a)-(b), it is clear that the resulting threads are able to concurrently execute instructions in different basic blocks of the original code, effectively following different control-flow paths. The potential of exploiting such parallelization opportunities is unique to a *global* multi-threaded scheduling, and constitutes its key advantage over *local* multi-threaded scheduling approaches.

3.5 Complexity Analysis

This subsection analyzes the complexity of GREMIO's partitioning algorithms. We first analyze the complexity of partitioning a single region with its inner loops coalesced, and then analyze the complexity of the hierarchical algorithm to handle loop nests. For the whole region's PDG, we denote n its number of nodes and e its number of arcs. By t we denote the target number of threads. Finally, we denote l the number of nodes in the HPDG tree, which is the number of loops in the region plus 1, and n_i and e_i the number of nodes and arcs in the HPDG for loop L_i, $0 \leq i \leq l$ ($i = 0$ for the whole region).

For a given loop L_i, the complexity of the DSC is $O(e_i + log(n_i))$ [32]. GREMIO's list scheduling, with checks for conflicts with currently scheduled nodes, has a complexity upper bound of $O(n_i^2)$.

In the dynamic programming algorithm, each node in the HPDG tree is processed exactly once. For each node, the clustering algorithm is applied once, and the list scheduling is applied t times, for each possible number of threads. Since the complexity of the clustering and list scheduling algorithms are more than linear, the worst case for the whole region's running time is when there are no loops. In this

Core	Functional Units: 6 issue, 6 ALU, 4 memory, 2 FP, 3 branch L1I Cache: 1 cycle, 16 KB, 4-way, 64B lines L1D Cache: 1 cycle, 16 KB, 4-way, 64B lines, write-through L2 Cache: 5,7,9 cycles, 256KB, 8-way, 128B lines, write-back Maximum Outstanding Loads: 16
Shared L3 Cache	$>$ 12 cycles, 1.5 MB, 12-way, 128B lines, write-back
Main Memory	Latency: 141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Figure 5. Machine details.

Benchmark	Function	Exec. %
adpcmdec	adpcm_decoder	100
adpcmenc	adpcm_coder	100
ks	FindMaxGpAndSwap	100
mpeg2enc	dist1	58
177.mesa	general_textured_triangle	32
179.art	match	49
181.mcf	refresh_potential	32
183.equake	smvp	63
188.ammp	mm_fv_update_nonbon	79
300.twolf	new_dbox_a	30
435.gromacs	inl1130	75
458.sjeng	std_eval	26

Figure 6. Selected benchmark functions.

case, there is a single node in the HPDG tree ($l = 1$), and $n_0 = n$ and $e_0 = e$. Therefore, the total complexity for the whole region is $O(e \times log(n) + t \times n^2)$. This low complexity enables this algorithm to be effectively applied in practice to regions with up to several thousands of instructions.

4 Evaluation

We implemented GREMIO in the VELOCITY compiler, a MT research compiler that targets Itanium 2. VELOCITY uses IMPACT's front-end to obtain an assembly-level IR. All traditional code optimizations, and some specific to Itanium 2, are performed in VELOCITY. GREMIO was performed after traditional optimizations, before the code is translated to Itanium 2's assembly, where Itanium 2-specific optimizations are performed, followed by register allocation and the final instruction scheduling pass.

To evaluate the performance of the code generated by VELOCITY, we used a validated cycle-accurate Itanium 2 processor performance model (IPC accurate to within 6% of real hardware for benchmarks measured [19]) to build a CMP model comprising two Itanium 2 cores connected by the *synchronization array* communication mechanism proposed in [21]. Figure 5 provides details about the simulator model, which was built using the Liberty Simulation Environment [30].

The synchronization array (SA) in the model works as a set of low-latency queues. In our implementation, there is a total of 256 queues, each one with 32 elements. The SA has a 1-cycle access latency and has four request ports that are shared between the two cores. The Itanium 2 ISA was extended with `produce` and `consume` instructions for inter-thread communication. These instructions use the

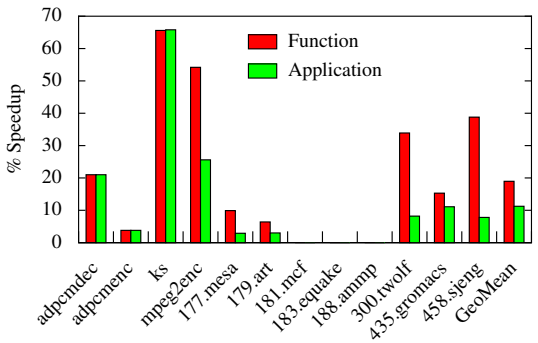


Figure 7. Speedup over single-threaded. (Note: no partition was chosen for three benchmarks.)

M pipeline, which is also used by memory instructions. This imposes the limit that only 4 of these instructions (minus any other memory instructions) can be issued per cycle on each core, since the Itanium 2 can issue only four M-type instructions in a given cycle. While the `consume` instructions can access the SA speculatively, the `produce` instructions write to the SA only on commit. As long as the SA queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles.

The highly-detailed nature of the validated Itanium 2 model prevented whole program simulation. Instead, detailed simulations were restricted to the functions in question in each benchmark. The execution was fast-forwarded through the remaining sections of the program while keeping the caches and branch predictors warm.

To demonstrate the potential of GREMIO, it was applied to important functions (at least 25% of the benchmark execution) of selected applications from the MediaBench, SPEC-CPU, and Pointer-Intensive benchmark suites. The benchmarks were restricted to those that currently go through our tool-chain, and for which our compiler statically estimated more than 10% speedup with either GREMIO or DSWP. Figure 6 lists the selected application functions along with their corresponding benchmark execution percentages.

Figure 7 presents the speedups for the selected benchmark functions. For each benchmark, the two bars illustrate the speedup on the selected function, as well as the corresponding speedup for the whole application. The overall speedup per function is 19.0% on average, with a maximum of 65.6% for *ks*. For three benchmarks, *181.mcf*, *183.equake*, and *188.ammp*, the compiler decided not to partition the selected function into multiple threads. The *458.sjeng* and *mpeg2enc* benchmarks strongly benefited from accumulator expansion. Several benchmarks benefited from communication optimizations currently being developed to improve the MT code generation algorithm used in this work.

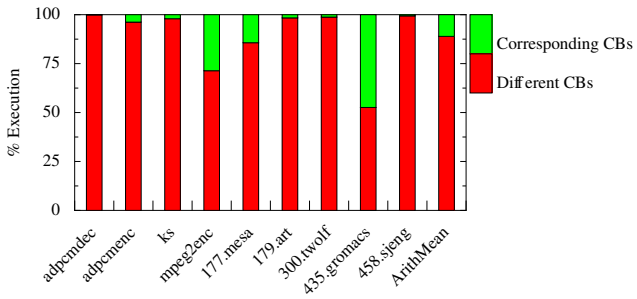


Figure 8. Percentage of execution on corresponding control blocks.

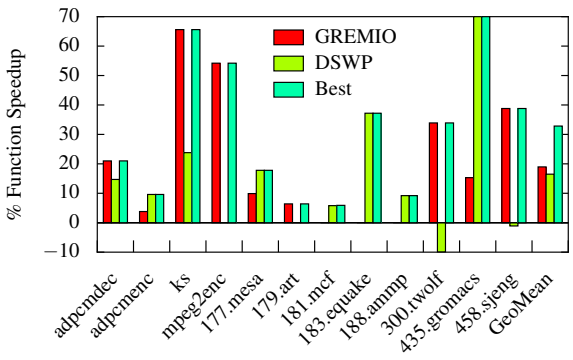


Figure 9. Speedups for GREMIO and DSWP over single-threaded execution.

Benchmark	Type of Parallelism
adpcmdec	CMT
adpcmenc	PMT
ks	PMT
mpeg2enc	CMT
177.mesa	CMT
179.art	CMT
300.twolf	PMT
435.gromacs	CMT
458.sjeng	PMT

Figure 10. Type of parallelism extracted by GREMIO.

4.1 Comparison to Local MT Scheduling

In order to verify the amount of parallelism obtained by GREMIO that can be extracted by LMT techniques, the multi-threaded execution traces were analyzed and the cycles classified in two categories. The first one corresponds to cycles in which both threads are executing instructions that belong to the same control block[2] in the original code. This is parallelism that can be extracted by LMT instruction scheduling techniques such as [14, 15, 22]. The remaining cycles correspond to the portion of the execution in which GMT instruction scheduling is necessary in order to expose the parallelism. Figure 8 illustrates the execution breakdown for the benchmarks parallelized by GREMIO. These results illustrate that, for the majority of these benchmarks, less than 2% of the parallelism obtained by GREMIO can be achieved by LMT techniques. The function in the SPEC-CPU 2006 FP *435.gromacs* benchmark, which contains two nested loops with no other control flow, is the only one in which a good fraction (47%) of the parallelism extracted by GREMIO is within control blocks.

4.2 Comparison to DSWP

Since the Decoupled Software Pipelining (DSWP) compilation technique proposed in [18] is a loop MT scheduling technique, it is natural to compare it with GREMIO. For this purpose, we also implemented DSWP in the VELOCITY compiler. Because DSWP is only applicable to loops, the compiler applies it to the most important outer loop in each of the selected benchmark functions. In fact, due to some peculiarities in our experimental infrastructure, GREMIO was also applied to the same outer loops in each function.

Figure 9 compares the function speedups achieved by GREMIO and DSWP. On average, DSWP achieves 16.5% speedup, compared to 19.0% for GREMIO. As can be seen, each of GREMIO and DSWP outperforms the other on half of the benchmarks. For *435.gromacs*, DSWP resulted in a 2.41× speedup, effectively benefiting from the doubled L2 cache capacity (the cores have private L2). A similar behavior was not observed with GREMIO because it unluckily kept the instructions responsible for most L2 misses in the same thread. Figure 9 also shows a bar for each benchmark indicating the speedup of the best performing version. This is the performance a compiler combining only these two MT techniques can ideally obtain. This best-of speedup averages 32.8%.

By the nature of DSWP, the parallelism it extracts is *Pipelined Multi-Threading* (PMT). In PMT, the communication among the threads inside the loop is unidirectional, and the parallelism extracted is generally across loop iterations. GREMIO, on the other hand, is not restricted to a specific kind of parallelism. Nevertheless, the algorithms described in this paper focus on hierarchically exploiting parallelism within loop iterations. In some cases, this can result in PMT, but it may also result in *Cyclic Multi-Threading* (CMT), with cyclic dependences among threads, or, theoretically, in *Independent Multi-Threading* (IMT), with no dependence among threads. In our experiments, GREMIO produced CMT for five benchmarks, and PMT for the other four. The table in Figure 10 shows the type of parallelism extracted by GREMIO for each benchmark. As can be noticed, CMT is superior to the PMT extracted by DSWP in one case (*adpcmdec*), and is also applicable in

[2] A control block, or extended basic block, is a sequence of instructions with a single entry (the first instruction) and potentially multiple exits.

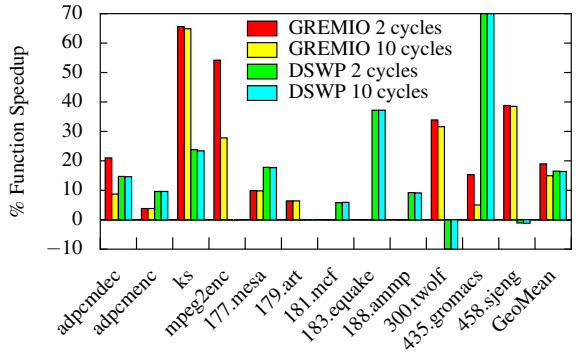


Figure 11. Speedup over single-threaded for different communication latencies.

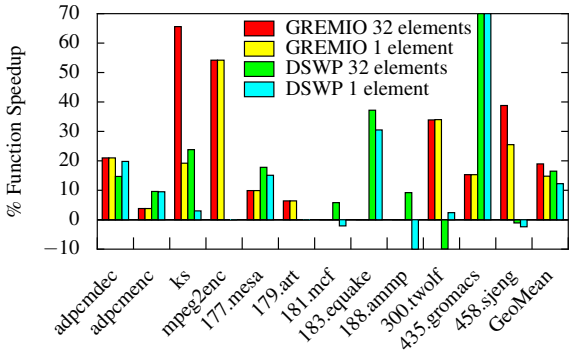


Figure 12. Speedup over single-threaded for different size of queues.

cases where DSWP is not (*mpeg2enc* and *179.art*). In other cases, DSWP outperforms the CMT extracted by GREMIO (*177.mesa* and *435.gromacs*). In the cases GREMIO extracted PMT, it is better than the PMT extracted by DSWP using the load-balancing heuristic described in [18] in some cases (*ks*, *300.twolf*, and *458.sjeng*).

4.3 Sensitivity Analysis: Communication Latency

Codes parallelized by DSWP have been shown to tolerate inter-core communication latencies [18]. This is due to the unidirectional nature of inter-thread communication in PMT. In order to assess the effect of communication latency for code generated by GREMIO, we conducted experiments with the inter-core communication latency increased from 2 cycles in our base model to 10 cycles. Figure 11 contains the results for both GREMIO and DSWP. The average speedup from GREMIO dropped from 19.0% to 14.9%, while DSWP is essentially unaffected. Not surprisingly, the GREMIO codes that are affected the most contain CMT-style parallelism (*adpcmdec* and *mpeg2enc*). However, not all benchmarks with CMT were slowed down by this increase in communication latency. In general, the CMT loops with small bodies are affected the most, since the communication latency represents a larger fraction of the loop body's execution.

4.4 Sensitivity Analysis: Queue Size

We also conducted experiments to measure how sensitive the parallelized codes are to the size of the communication queues. Figure 12 shows the resulting speedups, for both GREMIO and DSWP, on our base model, with 32-element queues, and with the size of the queues set to 1 element. The experiments show that most of the GREMIO codes are not affected. On the other hand, most of the DSWP codes are slowed down. The reason for this is that PMT can benefit from larger communication queues to improve the decoupling among the threads. On the other hand, for loops parallelized as CMT, one thread is never more than one loop iteration ahead of the other. As a result, single-entry queues are enough to obtain the maximum speedup in these cases. This means that a cheaper inter-core communication mechanism, with simple blocking registers, is enough to get the most parallelism out of CMT codes.

5 Related Work

There is a broad range of work on instruction scheduling. We briefly describe the *non-speculative* techniques most closely related to GREMIO, classifying them using a unified taxonomy that includes two orthogonal characteristics.

5.1 Scope of Scheduling

Instruction schedulers can be classified according to the scope of the regions they operate at a time. This classification includes: *local* techniques, operating at basic blocks or traces; *loop* techniques; and *global* techniques, operating at whole procedures. Classic examples of local scheduling include *local list scheduling* [16] and its extensions to traces [7] and superblocks [11]. Loop scheduling techniques include various sorts of software pipelining [12, 18]. The more general *global* techniques must be able to simultaneously schedule instructions from arbitrary CFG regions, potentially including the whole procedure. The special case of simultaneously scheduling instructions from control-equivalent basic blocks was studied in [1]. A more general approach, based on integer linear programming and combining scheduling and global code motion, was proposed in [31]. Compared to local approaches, global schedulers use a larger scope to help them making decisions, and thus have potential to obtain a better schedule. In addition to data dependences, schedulers operating beyond basic blocks must also preserve control dependences.

5.2 Single- versus Multi-Threaded Scheduling

Depending on the number of simultaneously executing threads they generate, scheduling techniques can be classified as either *single-threaded* or *multi-threaded*. Of course,

Num. of Threads	Scope			
	Basic Block	Trace	Loop	Procedure
Single	List Sched. [16]	Trace [3, 5, 7] Superblock [11]	SWP [12, 17]	GSTIS [1] ILP [31]
Multiple	Space-time [14] Convergent [15] DAE Sched. [22]		DSWP [18]	**GREMIO**

Table 1. Instruction scheduling space.

this characteristic is highly dependent on the target architecture. Single-threaded scheduling is commonly used for a wide range of single-threaded architectures, from simple RISC-like processors to very complex ones such as VLIW/EPIC [3, 12] and clustered architectures [5, 17].

Besides scheduling the original program's instructions (the *computation*), multi-threaded schedulers must also generate *communication* instructions to satisfy inter-thread dependences. For clustered single-threaded architectures, the scheduler also needs to insert communication instructions to move values from one register file to another. However, the fact that dependent instructions are executed in different threads makes the generation of communication more challenging for multi-threaded architectures.

Multi-threaded instruction scheduling techniques have been discussed earlier in the paper. The techniques proposed in the context of the RAW microprocessor [14, 15] are mostly based on *local multi-threaded* scheduling, using the so-called *asynchronous global branch* scheme of communicating branch directions at the end of basic blocks or traces. A similar approach is used by schedulers for decoupled access/execute architectures, which may even use specialized queues to communicate branch directions [22]. The DSWP technique, against which GREMIO was compared in our experiments, is a loop multi-threaded scheduling technique [18].

Table 1 summarizes how various existing scheduling techniques are classified according to our taxonomy. Horizontally, the more a techniques is to the right, the more general is its handling of control flow.

5.3 Comparison to MIMD Task Scheduling

Although operating at a different granularity, our work shares some similarities with task scheduling for parallel computers. Sarkar [24] describes general algorithms to partition and schedule functional parallel programs on multiprocessors. In particular, the idea of using a clustering prepass used here was inspired by Sarkar's work. However, our problem differs from his on a number of aspects, including the granularity of tasks and the abundance of parallelism in the source programs. Furthermore, our algorithms differ from his in many ways. For example we use list scheduling with a virtual schedule, which is very useful for the granularity of the parallelism we exploit, and our dynamic programming approach allows GREMIO to handle larger regions of code. Finally, our MT code generation algorithm is key to enable parallelization at the instruction granularity, by allowing multiple tasks to be assigned to a single thread.

6 Conclusion

This paper presented a general framework to extract non-speculative thread-level parallelism from global regions in general-purpose applications. Additionally, it described the algorithms used by GREMIO, a global multi-threaded scheduler built using this framework. GREMIO generalizes Decoupled Software Pipelining (DSWP) [18], by allowing the parallelization of non-loop regions and by not being restricted to pipeline parallelism. Using a fully automatic compiler implementation and a dual-core simulator built on top of validated Itanium 2 core models, GREMIO achieves an average of 19.0% speedup on important benchmark functions (maximum of 65.6%), translating to an average of 11.2% speedup over whole benchmarks (maximum of 65.6%). Our experiments also showed that a compiler with both DSWP and GREMIO can achieve an average of 32.8% speedup on the same benchmark functions, compared to 16.5% with only DSWP. In addition, when targeting more threads, we conjecture that GREMIO and DSWP can be synergistically combined. For example, GREMIO can be used to speed up the slowest stage in a DSWPed loop, or DSWP can be used to obtain a better parallelization for inner loops of a region to which GREMIO is applied. Exploring these opportunities is the subject of future work.

Acknowledgments

We thank the entire Liberty Research Group and Vivek Sarkar for their feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has been supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

References

[1] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[2] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling

compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, April 1987.

[4] 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*, 13(4):451–490, October 1991.

[5] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1985.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY, 1979.

[9] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[10] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 146–157, 1988.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[12] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[13] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[14] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw Machine. In *Proceedings of the Eighth International Confrence on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[15] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002.

[16] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[17] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 103–114, December 1998.

[18] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[19] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[20] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 259–269, December 2006.

[21] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[22] K. Rich and M. Farrens. Code partitioning in decoupled compilers. In *Proceedings of the 6th European Conference on Parallel Processing*, pages 1008–1017, Munich, Germany, September 2000.

[23] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[24] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, MA, 1989.

[25] V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[26] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu. Field-testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2004.

[27] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[28] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[29] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[30] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[31] S. Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[32] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.