Global Multi-Threaded Instruction Scheduling: Technique and Initial Results

Guilherme Ottoni David I. August

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

Recently, the microprocessor industry has reached hard physical and micro-architectural limits that have prevented the continuous clock-rate increase, which had been the major source of performance gains for decades. These impediments, in conjunction with the still increasing transistor counts per chip, have driven all major microprocessor manufacturers toward Chip Multiprocessor (CMP) designs. Although CMPs are able to concurrently pursue multiple threads of execution, they do not directly improve the performance of most applications, which are written in sequential languages. In effect, the move to CMPs has shifted even more the task of improving performance from the hardware to the software. In order to support more effective parallelization of sequential applications, computer architects have proposed CMPs with light-weight communication mechanisms [26, 24, 22]. Despite such support, proposed multi-threaded scheduling techniques have generally demonstrated little effectiveness [15, 16] in extracting parallelism from general-purpose, sequential applications. We call these techniques *local multi-threaded scheduling*, because they basically exploit parallelism within straight-line regions of code. A key observation of this paper is that local multi-threaded techniques do not exploit the main feature of CMPs: the ability to concurrently execute instructions from different control-flow regions. In order to benefit from this powerful CMP characteristic, it is necessary to perform *global multi-threaded scheduling*, which simultaneously schedules instructions from different basic blocks to enable their concurrent execution. This paper presents algorithms to perform global scheduling for communication-exposed multi-threaded architectures. By *global* we mean that our technique simultaneously schedules instructions from an arbitrary code region. Very encouraging preliminary results, targeting a dual-core Itanium 2 model, are presented for selected benchmark applications.

1 Introduction

In the last few years, the microprocessor industry has been undergoing what is being considered one of its major changes. 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 on parallel applications. Although ideally programmers could rewrite all the 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 more viable alternative is 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 have remarkably regular array-based memory accesses and little control flow.

Because the parallelism available in non-scientific applications is typically much more fine-grained, computer architects have proposed simple hardware support mechanisms to enable light-weight fine-grained communication [26, 24, 22, 21], generally called *scalar operand networks*. These mechanisms typically consist of an on-chip interconnect between the processor cores, and special `produce` and `consume` instructions to communicate scalar values from one core to another. To the software, these communication mechanisms look like sets of hardware-implemented queues. Extracting parallelism for these processors consists of partitioning the computation into threads, to execute on different cores, and inserting communication instructions to satisfy inter-thread dependences. The parallelism exposed by these processors is of finer granularity than what is typically exploited by programmers in parallel systems, making it even harder to manually explore these opportunities. Therefore, generating code that exploits these opportunities is better performed by a compiler's instruction scheduler.

Instruction scheduling techniques can be classified as either *local* or *global*. While local techniques schedule the instructions of each basic block independently, global approaches simultaneously consider instructions from different basic blocks. Most of the existing multi-threaded scheduling techniques are based on local scheduling [15, 16]. We call these techniques *local multi-threaded* (LMT) scheduling.

One of our key observations is that LMT scheduling techniques do not exploit the main advantage brought by multi-threaded architectures: the ability to simultaneously follow different execution paths in different processor cores. Given the typically small size of basic blocks in general-purpose applications, we believe it is crucial to exploit parallelism beyond basic block boundaries in order to successfully extract parallelism from these applications. As an example, consider the sample C code in Figure 1. Although these loops may iterate for a large number of iterations, very little instruction-level parallelism is available within each individual basic block. For such control-intensive codes, any local scheduling technique will hardly extract any thread-level 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, i.e. to simultaneously consider instructions from different basic blocks during scheduling. The major complication of going from any local analysis or optimization to its corresponding global version is the presence of control flow. In this paper, we demonstrate how control flow can effectively be handled to enable GMT scheduling. Our technique combines a global multi-threaded list scheduling heuristic with a novel global multi-threaded code generation algorithm. These algorithms are based on a *Program Dependence Graph* (PDG) representation [6], which includes both data and control dependences.

Overall, this paper makes the following contributions:

1. It introduces the concept of global multi-threaded (GMT) scheduling, which we believe is key to fully take advantage of multi-threaded architectures, in particular to parallelize general-purpose applications.
2. It shows how to handle control flow in order to enable GMT scheduling, and presents a novel global multi-threaded list scheduling heuristic.
3. It presents an effective dynamic programming algorithm to efficiently perform GMT scheduling on large code regions composed of complex loop nests.
4. It presents a general algorithm to generate multi-threaded code from arbitrary partitions of the instructions among the threads. This algorithm is a generalization for arbitrary CFGs of the algorithm used for loop scheduling in [19], and it can be used with *any* GMT scheduling heuristic.

```
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.

5. It shows initial promising 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 PDGs. We present our GMT scheduling heuristics in Section 3, followed by our multi-threaded code generation algorithm in Section 4. In Section 5, we present experimental results. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 Program Dependence Graphs

Local scheduling techniques operate by constructing a data dependence graph representing all data dependences that must be respected in order to keep the original program's semantics. 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 [13]: *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.

The key addition from a local scheduling 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. A control dependence arc from a branch instruction X to an instruction Y means that, depending on the direction taken at X, Y either must or may not be executed. Dependence graphs including both data and control dependences are generally called *Program Dependence Graphs* (PDGs) [6]. Cytron et al. [4] proposed an efficient algorithm to compute control dependences for arbitrary CFGs, based on *post-dominance frontiers*.

```
(A)  B1: move   r1 = 0         ;; r1 contains s1
(B)      move   r2 = 0         ;; r2 contains s2
(C)      load   r3 = [head]    ;; r3 contains p
(D)  B2: branch r3 == 0, B4
(E)  B3: load   r4 = [r3]      ;; load p->value
(F)      add    r1 = r1, r4
(G)      load   r3 = [r3+4]    ;; load p->next
(H)      jump   B2
(I)  B4: move   r5 = @a        ;; r5 pts. to a[i]
(J)  B5: load   r6 = [r5]      ;; load a[i]
(K)      branch r6 == 0, B7
(L)  B6: add    r2 = r2, r6
(M)      add    r5 = r5, 4
(N)      jump   B5
(O)  B7: mult   r7 = r1, r1
(P)      div    r8 = r7, r2
```

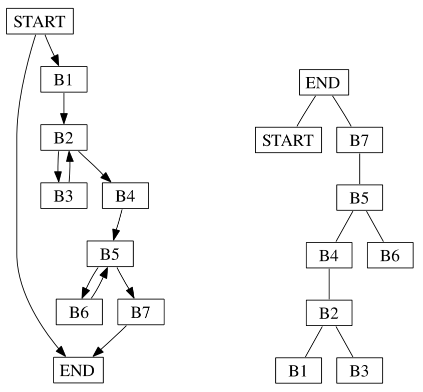
Figure 2. Low-level IR for the code in Figure 1.

2.1 PDG for Global Multi-Threaded Scheduling

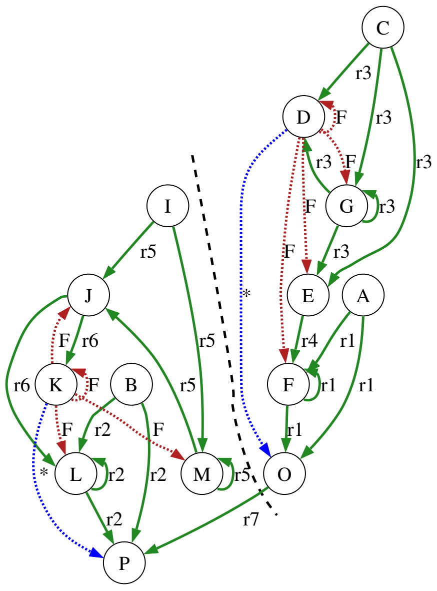
The PDG for global multi-threaded (GMT) scheduling contains one vertex for each instruction in the code, with the exception of `jump` instructions. These instructions are left aside because their effect is embedded in the control dependences. Besides this, they only serve to put the basic blocks of the CFG into a linear representation.

We now precisely define the PDG dependence arcs necessary for our GMT scheduling. We denote V_G and E_G, respectively, the sets of vertices and arcs of a graph G.

- Register data dependences: only flow dependences through registers need to be considered, and anti- and output dependences can be ignored. The reason for this is that, if two instructions involved in an anti- or output dependence are scheduled on different threads, they will execute in two processors with different register sets. In other words, the use of different register sets automatically eliminates false dependences. Additionally, for instructions scheduled on the same thread, our algorithm preserves their order, thus naturally respecting intra-thread dependences. A register dependence from instruction X to Y involving r_i is denoted $X \rightarrow^{r_i} Y$.
- Memory data dependences: for memory, all flow, anti- and output dependences need to be taken into account, as the memory store is shared by the threads. Memory dependences are denoted $X \rightarrow^M Y$.
- Direct control dependences: our PDG includes control dependences. We denote $X \rightarrow^T Y$ for taken and $X \rightarrow^F Y$ for not-taken branch directions.
- Transitive control dependences: for each dependence $(X \rightarrow Y) \in E_{PDG}$, and for each branch instruction B on which X is dependent, a transitive control dependence arc $B \rightarrow^* Y$ is added to E_{PDG}. The reason for these dependences will become evident in Section 4, when we describe our multi-threaded code generation algorithm.



(a) CFG (b) Post-dom. Tree



(c) PDG

Figure 3. (a) CFG, (b) post-dominance tree, and (c) PDG.

Figure 2 illustrates a low-level representation for the code in Figure 1, and Figure 3 contains the corresponding CFG, post-dominance tree, and PDG.

Using the PDG constructed as described above, different GMT scheduling heuristics can be applied to choose a global schedule, i.e. a partition of the instructions among the threads. For example, for the PDG in the example of Figure 3(c), a heuristic may decide to partition the code in two threads as depicted by the dashed line. This partition corresponds to scheduling each loop of Figure 1 onto a separate thread.

3 Global Multi-Threaded Instruction Scheduling

In this section, we describe in details our GMT scheduling algorithms, and illustrate them on the example of Figure 1. This section is only concerned with the thread-level scheduling decisions, i.e. mapping the instructions of the original sequential program onto threads. The multi-threaded code generation algorithm is presented in Section 4. Because each thread generated by our technique is intended to execute on a different core, we interchangeably say that an instruction is scheduled on a thread or core.

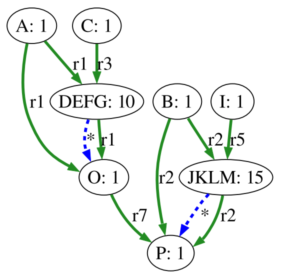
Although a multi-threaded instruction scheduler can be combined with a traditional single-threaded scheduler, we opted not to do so in this work. One reason for this is that the multi-threaded code generated by our technique can be further optimized before the actual assembly code generation. Additionally, exposing all the machine details to the GMT scheduler would make its implementation more complex. Instead, we preferred to keep our GMT scheduler simpler by providing it with just a few key characteristics of the target processor, namely the number of cores and the issue-width of each core. A latency of one cycle is assumed for most instructions (except for function calls), and no information about structural hazards is used.

Our GMT scheduler uses a PDG as intermediate representation for both scheduling decisions and code generation. Because our technique is global, targeting arbitrary code regions, it must deal with the possibility of cycles in a PDG. Scheduling of cyclic graphs is more complex than scheduling of acyclic graphs. 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 time linear, through a topological sort. For cyclic graphs, however, even finding the longest path is NP-hard [8].

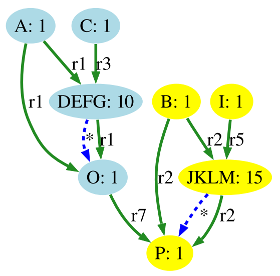
Given the inherent difficulty of the global scheduling problem for cyclic code regions, we use a simplifying approach that reduces it to the acyclic scheduling problem, for which well-known heuristics based on list scheduling [9] exist. In order to reduce the cyclic scheduling problem to an acyclic one, we make 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. Secondly, if the code region being scheduled is a loop, all the loop-carried dependences are disregarded. To deal with the possibility of irreducible code, we use a loop hierarchy that includes irreducible loops [10]. It is important to notice that these simplifying assumptions are used for scheduling decisions only; our 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 4(a) illustrates the HPDG corresponding to the PDG from Figure 3(c). The nodes are labeled by their corresponding nodes in the PDG, followed by their estimated latency. There are only two loop nodes in this example: DEFG and JKLM.

Even after eliminating the cycles in the PDG, control flow still poses additional complications to GMT instruction scheduling that do not exist for local scheduling. In local scheduling, there is a guarantee that all instructions will execute, i.e. 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



(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) Schedule

Figure 4. (a) HPDG for the PDG from Figure 3(c). (b) Clustered HPDG. (c) Chosen Schedule.

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 anymore. For example, the fact that instruction A executes may not be related to the execution of B, or may even imply that B will not execute. To deal with the different possibilities, we introduce three different *control relations* among instructions, which are used during our scheduling algorithms.

Definition 1 (Control Relations) *Given two HPDG nodes A and B, we call them:*

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

To illustrate these relations, consider the HPDG from Figure 4(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.

Another problem intrinsic to GMT scheduling is that the multiple threads generated will execute on different cores, and so it is necessary to take the communication overhead into account while making scheduling decisions. For example, even though two instructions can be issued in parallel on different threads, this might not be profitable due to necessary communication to move their operands from one core to another. To address this problem, we use a *clustering* pre-scheduling pass on the HPDG, which takes into account the inter-core 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 we use, and our multi-threaded instruction scheduling heuristic is described in Section 3.2.

3.1 Clustering Algorithm

Our clustering algorithm is an adaptation of the *Dominant Sequence Clustering (DSC)* algorithm [29], widely used for task scheduling in parallel computing. Here we briefly describe DSC, and point out the modifications we incorporated to more adequately deal with powerful ILP processor cores.

The DSC algorithm, like all multi-processor task scheduling algorithms, performs clustering on a directed acyclic graph (DAG). Therefore, because of the simplifications we performed to reduce our cyclic scheduling problem into an acyclic one, we can rely on previous research on multi-processor task scheduling. We chose to use DSC because it has been shown to be both effective and efficient, being able to handle graphs with thousands of nodes [29]. Efficiency is important for instruction scheduling because of the potentially huge number of nodes in a HPDG.

The DSC algorithm assumes that each cluster will be executed on a different processor (core for us). The later scheduling pass may schedule multiple clusters on the same thread to cope with a smaller number of processors.

DSC starts by assigning each instruction to a different 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 instructions 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 cluster increase the critical path, this node is left alone in its own cluster.

In this work, we use a slight modification of the DSC algorithm to deal with the ILP power available in modern processor cores. To do that, we do not assume a sequential execution inside each cluster. Instead, we assume the node being merged will be issued at the earliest cycle such that: (a) its inter-cluster dependences are satisfied (including the communication cost), (b) its intra-cluster dependences are fulfilled, (c) its *control conflicting* nodes inside the cluster are finished, and (d) there is an issue slot available.

In addition, in our DSC variation, we use a more refined breakdown of the communication overhead components, which suits better the inter-core communication mechanism we assume. Whenever there is an inter-cluster dependence from instruction A to instruction B, we assume the following communication latencies:

- *Producing Latency* in A's cluster, after A executes.
- *Consuming Latency* in B's cluster, before B executes.
- *Communication Latency*, which is added to A's finish cycle to estimate when A's value will be available for use by B.

Finally, in our DSC implementation, we perform a post-pass to eliminate some trivial clusters. Specifically, we look for all clusters that contain a single simple node and that has dependences with instructions from a single cluster. When such a trivial cluster is found, it is merged with its single adjacent cluster.

Figure 4(b) illustrates the clusters resulting from this algorithm. For simplicity, we assume here that the producing, consuming, and communication latencies are all one cycle. Initially, each node is its own cluster. The first nodes to be processed are B and L, which have priority (i.e. the length of the longest, critical path through it) equal to 23. For example, for B, the longest path includes 17 cycles of execution latency, plus 2 cycles of producing latency (for B and JKLM), 2 cycles of communication latency (for arcs B→JKLM and JKLM→P), and 2 cycles of consuming latency (for both JKLM and P). As neither B nor I have predecessors, they are left on their own clusters. Next, node JKLM is processed, which also has priority 23. Merging this node with any of its predecessors will not increase its priority, so we arbitrarily merge it with B. After that, nodes A and C are processed, and each remains in its own cluster. Notice that, even though node P has higher priority, it does not have all its predecessors processed yet. Next, node DEFG is processed and, similarly to what happened to JKLM, it is arbitrarily merged with one of its predecessors, A. At this point, node O is processed, and it is merged with its only predecessor cluster, which contains A and DEFG. Finally, P is processed, and it is merged with the cluster containing B and JKLM, what reduces the critical path to 20. At this point, the trivial-cluster elimination post-pass is performed, and two trivial clusters are merged: (1) node C is merged with the cluster containing A, DEFG, and O; and (2) node I is merged with the cluster containing B, JKLM, and P.

3.2 Global Multi-Threaded List Scheduling

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

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.

For traditional, single-threaded instruction scheduling, the resources correspond to the processor's functional units. For GMT instruction scheduling, 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 opportunities available in a *global* scheduling problem, in particular 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, we simply schedule it on the thread previously chosen for this cluster.

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

1. *Startup Overhead*: this is the difference between the first cycle in which the node in consideration can be scheduled on the given thread and the current cycle.
2. *Communication Overhead*: this is the total number of cycles that will be necessary to execute all `produce` and `consume` instructions to satisfy dependences between this cluster and instructions in clusters already scheduled on different threads.
3. *Resource-Conflict Overhead*: this is an 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 already assigned to this thread. This takes into account both the average resource utilization per cycle for the unfinished instructions, as well as the total latency of the current cluster. In effect, this overhead is more important for larger clusters, and it is useful to improve the load balance among threads.
4. *Control-Conflict Overhead*: this is an estimated number of cycles in which instructions in this cluster will not be able to execute in this thread due to control conflicts with unfinished instructions of other clusters already scheduled on this thread. To compute this estimate, we use the latency of the unfinished instructions in other clusters assigned to this thread, weighted by the probability that a control conflict will impede the issue of each instruction in the cluster being scheduled. This control conflict probability is computed as the latency of the unfinished instructions that are con-

trol conflicting with the one being considered, over the total latency of all unfinished instructions.

Once we choose the thread in which a HPDG node is to be scheduled, it is necessary to estimate the cycle in which that node can be issued in this core. Although we do not perform the actual scheduling at this point, this estimate is used to guide the GMT scheduling for the remaining nodes.

In order to find the estimated cycle in which a node can be issued in the chosen thread, it is necessary to consider two restrictions. First, we need 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, as 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, however, for target cores that support predicated execution, 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 our 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 4(c)). The list scheduling algorithm processes the nodes in the clustered HPDG (Figure 4(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 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, we may schedule nodes A, C, or JKLM. 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 startup, control-conflict, and resource-conflict overheads. Therefore, core 0 is chosen for node A. The algorithm then proceeds, and the remaining scheduling decisions are all cluster-based. Figure 4(c) illustrates the final schedule built and the partitioning of the instructions among the threads.

3.3 Handling Loop Nests

Although our scheduling algorithm follows the clusters formed a priori, we make an exception 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.

We handle inner loops as follows. For now, assume that we have an estimate for the latency to execute one invocation of an inner loop L_j using a number of cores i from 1 up to the number N of cores on the target processor. Let $latency_{L_j}(i)$, $1 \leq i \leq N$, denote these latencies. Considering L_j's control conflicts, we compute the cycle in which each core will finish executing L_j's control-conflicting nodes already scheduled on it. From that, we can compute the earliest cycle in which a given number of cores i will be available for L_j, denoted by $cycle\_available_{L_j}(i)$, $1 \leq i \leq N$. With that, we choose to schedule this loop node on a number of cores k 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 cores available and the benefit from executing the loop node on more threads. If more than k threads are available at $cycle\_available(k)$ (e.g. all threads will be available in the same cycle, but we do not need all of them), then we pick 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 do we compute the $latency_{L_j}(i)$ for each child loop L_j in the HPDG? Intuitively, this is a recursive question, since what we have been doing is scheduling a code region on multiple threads, with the goal of minimizing its execution latency. This naturally leads to a recursive solution. But even better, we can apply dynamic programming to efficiently solve this problem in polynomial time. In addition, were our list scheduling algorithm perfect, this would be able to compute the *optimal* scheduling for an arbitrary code region.

More specifically, our dynamic programming solution works as follows. First, we compute the loop hierarchy for the region we want to schedule. This can be viewed as a loop tree, where the root represents the whole region (in case the region is not a loop itself). 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. This 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, we choose 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 the minimum. 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 $latency_{L_j}(i)$.

As a final note, we point 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 [19], besides the GMT list scheduling described here. The evaluation of such general framework is beyond the scope of this paper.

4 Multi-Threaded Code Generation

We now describe our Multi-Threaded Code Generation (MT-CG) algorithm. For any global schedule chosen, this algorithm generates corresponding multi-threaded code, automatically inserting the communication and synchronization instructions necessary to preserve the program's dependences.

Figure 5 presents the MT-CG algorithm, which takes as input the original control-flow graph (*CFG*), the *PDG* constructed as described in Section 2.1, and the chosen global schedule (*GS*). As output, this algorithm produces a new CFG for each of the resulting threads, containing its corresponding instructions, and including the necessary communication and synchronization instructions.

In essence, the MT-CG algorithm works as follows. For each of the threads specified by the global schedule, it generates a new CFG with only the necessary basic blocks for this thread. Then, the instructions are inserted in the thread to which they were scheduled. After that, 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.

Before going into the details of the algorithm in Figure 5, let us introduce the notation used. P_i denotes a partition (thread) in *GS*, and CFG_i denotes its corresponding control-flow graph. For a given instruction I, $bb(I)$ is the basic block containing I in the CFG, and $point_j(I)$ is the point in CFG_j corresponding to the location of I in the CFG.

The first step of the algorithm is to find the set of the *relevant basic blocks* for each partition (thread) P_i in *GS*. The set of relevant basic blocks for P_i contains the set of blocks that will compose CFG_i. Additionally, CFG_i is carefully constructed so that each of its basic blocks has exactly the same condition of execution as its corresponding block in CFG. The procedure Relevant_BBs, lines 26-31 in Figure 5, describes how to compute such set of basic blocks. This set contains one block for each block in the original CFG that contains either (a) an instruction scheduled to P_i, or (b) an instruction on which any of P_i's instructions depends (i.e. a source of a dependence with an instruction in P_i as the target). The reason for including basic blocks containing instructions in P_i is obvious, as they will hold these instructions in the generated code. The reason for adding the basic blocks containing instructions on which P_i's instructions depend is related to a property used to preserve the semantics in the transformed code: inter-thread communication instructions are inserted at the point of the *source instruction*, so as to keep the exact condition under which this dependence happens. Notice that this particular choice of where to communicate a dependence is somewhat arbitrary, and may not be optimal. However, this choice does simplify the proof of correctness of the algorithm. This choice of where dependences are communicated is also the motive for making the transitive control dependence arcs explicit in the PDG: if these dependence arcs connect instructions scheduled to different threads, these branches need to be communicated so that the inter-thread dependences keep their condition of execution. The call to *create_corresp_bb*$_i(B)$ (line 28) creates the block B_i corresponding to B in CFG_i. The mappings between B and B_i are denoted by: *corresp_bb*$_i(B)$ = B_i, and *orig_bb*(B_i) = B.

The next step of the MT_CG algorithm (lines 3-5) is to insert the instructions in P_i into their corresponding basic blocks in CFG_i. The instructions are inserted in the same relative order as in the original code, so that intra-thread dependences are naturally satisfied. After this, the code in lines 6-18 inserts communication and synchronization instructions in order to preserve the inter-thread dependences. For each such dependence, a separate communication queue is used[1]. Notice that, as mentioned above, the communication instructions are always inserted at the point corresponding to the instruction that is the source of the dependence. The actual communication instructions inserted depend on the type of the dependence. Register dependences are implemented by communicating the register in question right after the point that it is produced in the source thread. For memory dependences, purely synchronization instructions are inserted to enforce that their relative order of execution is preserved. Finally, control dependences are more involving. In the source thread, before the branch is executed, its register operand is sent. In the thread that is the sink of the dependence, a `consume` instruction is inserted to get the corresponding register value, and then an equivalent branch instruction is inserted to mimic the same control behavior.

[1]A separate queue is used just for simplicity. Later, a queue-allocation algorithm can reduce the number of queues necessary.

```
      MT_CG ($CFG$, $PDG$, $GS$)
(1)    for each $P_i \in GS$ do
(2)      $V_{CFG_i} \leftarrow$ Relevant_BBs($CFG$, $PDG$, $P_i$)
(3)    for each $I \in V_{PDG}$, in original program order, do
(4)      let $i$ be such that $I \in P_i$
(5)      add_last($corresp\_bb_i(bb(I))$, $I$)
(6)    for each $(I \to J) \in E_{PDG}$, where $I \in P_i$, $J \in P_j$, $P_i \neq P_j$ do
(7)      $q \leftarrow get\_free\_queue()$
(8)      if $dep\_type(I \to J) = r_k$ then
(9)        add_after($corresp\_bb_i(bb(I)), I$, "produce [$q$] = $r_k$")
(10)       add_before($corresp\_bb_j(bb(I)), point_j(I)$, "consume $r_k$ = [$q$]")
(11)     else if $dep\_type(I \to J) = M$ then
(12)       add_after($corresp\_bb_i(bb(I)), I$, "produce [$q$]")
(13)       add_before($corresp\_bb_j(bb(I)), point_j(I)$, "consume [$q$]")
(14)     else    //  control dependence
(15)       $r_k \leftarrow register\_argument(I)$
(16)       add_before($corresp\_bb_i(bb(I)), I$, "produce [$q$] = $r_k$")
(17)       add_before($corresp\_bb_j(bb(I)), point_j(I)$, "consume $r_k$ = [$q$]")
(18)       add_before($corresp\_bb_j(bb(I)), point_j(I), I$)
(19)   for each $P_i \in GS$ do
(20)     add $START$ and $END$ nodes to $CFG_i$
(21)     for each branch $I \in P_i$ do
(22)       redirect_target($I$, $closest\_relevant\_postdom_i(target(I))$)
(23)     for each $B \in V_{CFG_i}$ do
(24)       CRS $\leftarrow closest\_relevant\_postdom_i(succ(orig\_bb(B)))$
(25)       add_last($B$, "jump CRS")

      Relevant_BBs($CFG$, $PDG$, $P_i$)
(26)   $RB \leftarrow \emptyset$
(27)   for each $I \in P_i$ do
(28)     $RB \leftarrow RB \cup \{create\_corresp\_bb_i(bb(I))\}$
(29)     for each $J \mid (J \to I) \in E_{PDG}$ do
(30)       $RB \leftarrow RB \cup \{bb(J)\}$
(31)   return $RB$
```

Figure 5. Multi-threaded code generation algorithm.

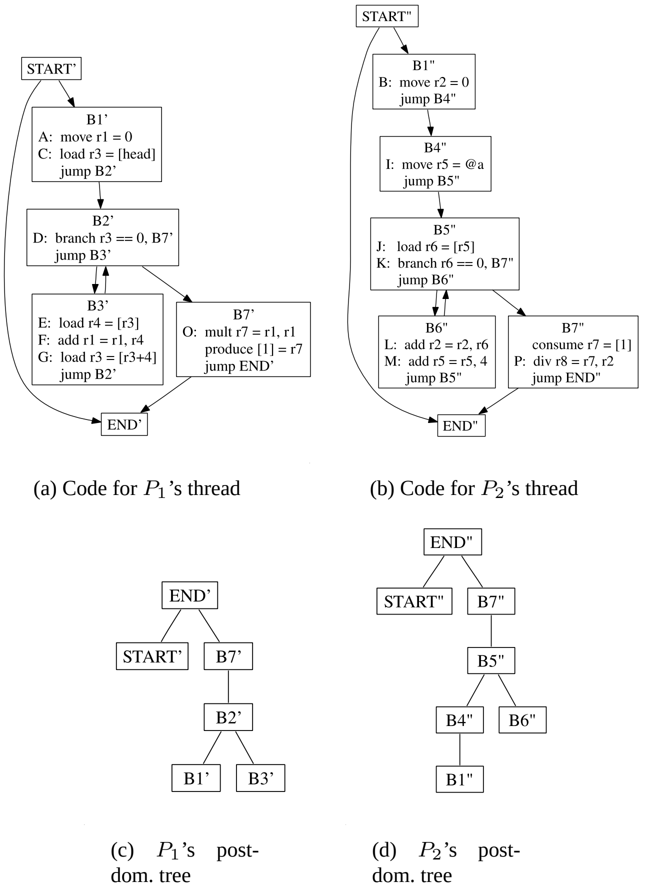


Figure 6. Resulting multi-threaded code and corresponding post-dominance trees.

The last step of the algorithm (lines 19-25) is to insert *START* and *END* nodes in the new CFGs, and to fix the branch targets and insert `jump` instructions to properly connect the basic blocks in each of the new CFGs. Because not all the basic blocks in the original CFG have a corresponding one in each new CFG, finding the adequate branch/jump targets is non-trivial. In order to preserve the control dependences, the branch/jump targets need to be retargeted to the *closest post-dominator basic block B* of the original target/successor, in the original CFG, such that *B* is *relevant* to the new CFG. We call such block *B* the *closest relevant post-dominator*, in the new CFG, of the original target/successor. Notice that such post-dominator basic block always exists as every vertex is post-dominated by *END*, which is relevant to every CFG.

A simple optimization, not illustrated in the algorithm in Figure 5, is that dependences between the same pair of threads that have the same source instruction need only to be communicated once. Moreover, many jump instructions inserted to connect the blocks in each new CFG can be eliminated by code layout and jump optimizations.

We have proved that our MT_CG algorithm preserves all the dependences in the PDG. Combined with Sarkar's result showing that any transformation that preserves all dependences in the PDG also preserves the program's semantics [25], this leads to the correctness proof of the MT_CG algorithm. In interest of space, we omit these proofs.

Figures 6(a)-(b) illustrate the generated code for the two threads corresponding to the global schedule depicted in Figure 3(c). In Figures 6(c)-(d), the post-dominator trees for the new CFGs are illustrated. As can be easily checked, 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 `produce` and `consume` instructions, corresponding to the only cross-thread dependence in the schedule shown in Figure 3(c).

By analyzing the resulting code in Figures 6(a)-(b), it should be clear that the resulting threads are able to concurrently execute instructions in different basic blocks of the original code, effectively following different control-flow

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

Table 1. Machine details.

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.

5 Evaluation

We implemented our GMT scheduling technique in the Velocity compiler, a research compiler derived from UIUC's IMPACT compiler [1] that targets Itanium 2. Velocity uses IMPACT's front-end, and the resulting IMPACT's Lcode is then translated into Velocity's X code IR. All traditional code optimizations are performed in Velocity, as well as some Itanium 2 specific optimizations. Our GMT list scheduling 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. Velocity is parameterized with respect to the number of cores in the target processor.

To evaluate the performance of the code generated by Velocity, we used a validated cycle-accurate Itanium 2 processor [12] performance model (IPC accurate to within 6% of real hardware for benchmarks measured [20]) to build a CMP model comprising two Itanium 2 cores connected by the *synchronization array* communication mechanism proposed in [22]. Table 1 provides details about the simulator model. The simulator was built using the Liberty Simulation Environment [27].

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 IA-64 ISA was extended with `produce` and `consume` instructions for inter-thread communication. These instructions use the 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

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
300.twolf	new_dbox_a	30
435.gromacs	inl1130	75

Table 2. Selected benchmark functions.

model prevented whole program simulation. Instead, detailed simulations were restricted to the functions in question in each benchmark. We fast-forwarded through the remaining sections of the program while keeping the caches and branch predictors warm.

To demonstrate the potential of our GMT scheduling technique, we applied it to important functions of selected applications from the MediaBench, SPEC-CPU, and Pointer-Intensive benchmark suites. Table 2 lists the selected application functions along with their corresponding benchmark execution percentages.

Figure 7 presents the speedup for the selected benchmark functions. For each benchmark, the two bars illustrate the speedup achieved with 2 and 10 cycles for the inter-core communication latency. With a 2-cycle communication latency, the speedups vary from 3.8% for adpcmenc to 173.3% for 435.gromacs, with a geometric mean of 38.7%.

The 2.7x speedup on two threads for 435.gromacs came as a surprise. The doubly nested loops in function `inl1130` contains an enormous amount of floating-point operations. We verified that, in the single-threaded version, this code suffers from a large number of spills of floating-point registers during register allocation (using graph coloring). In the multi-threaded version, the availability of twice as many registers enabled a much smaller number of spills, thus resulting in a reduced schedule height. Even though we just observed this advantage in one benchmark, we believe that this usage of additional resources will be even more beneficial in CMPs with smaller cores.

The results in Figure 7 show that, increasing the communication latency from 2 to 10 cycles, the geometric mean of the speedup drops from 38.7% to 31.8%. Additionally, we notice that the sensitiveness to the increased communication latency varies form benchmark to benchmark. In general, we noticed that functions with outer loops that iterate very fast (i.e. with small loop bodies), such the ones from adpcmdec and mpeg2enc, are more affected by the increased inter-core communication latencies. This is because, for such smaller loops, the communication latency corresponds to a larger percentage of the time necessary to execute one iteration of the loop.

We also conducted experiments to measure how sensitive the parallelized codes are to the size of the communication queues. Figure 8 shows the resulting speedups on our base model, with 32-element queues, and with the size

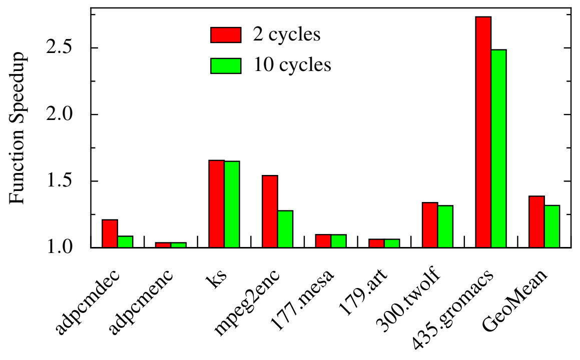


Figure 7. Speedup over single-threaded, for different inter-core communication latencies.

of the queues set to 1 element. The experiments show that only one of the benchmarks, ks, is affected by the reduced queue sizes. Investigation of the generated codes showed that, although the algorithms presented here may generate acyclic multi-threading such as DSWP [19], this was not the case in general. In fact, this was only observed for one inner loop in the ks benchmark. All other generated codes have cyclic multi-threading, in which pairs of cyclically dependent threads will always be less than one iteration apart. This explains why the benchmarks parallelized here are more susceptible to longer inter-thread latencies than the ones generated by DSWP [19]. The good side of this is that a cheaper inter-core communication mechanism, with simple blocking registers, is enough here.

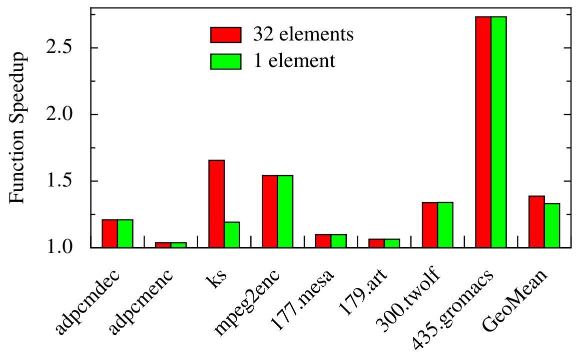


Figure 8. Speedup over single-threaded, for different communication queue sizes.

6 Related Work

There is a broad range of related work on instruction scheduling. In this section, we briefly describe and contrast the techniques mostly related to ours. The techniques are classified using a unified taxonomy that includes two orthogonal characteristics.

6.1 Local *versus* Global Scheduling

Instruction schedulers can be classified as either *local* or *global*. *Local* techniques independently schedule each straight-line sequence of instructions, typically a basic block. For each basic block, the instructions are scheduled respecting a *Data Dependence Graph* (DDG), where each vertex corresponds to an instruction and the arcs determine a partial ordering that must be respected in order to keep the correct program behavior. A classic example of local scheduling is *local list scheduling* [17], used in many optimizing compilers.

On the other hand, global schedulers simultaneously consider instructions from different basic blocks when making their schedule decisions. The set of basic blocks scheduled simultaneously can have different characteristics. For example, some techniques consider only instructions in basic blocks that form a trace in the CFG [7, 11]. Others schedule all the instructions in a set of basic blocks that form a loop in the program [14, 19]. The more general techniques must be able to simultaneously scheduled instructions in arbitrary CFG regions, potentially including the whole procedure. The special case of simultaneously scheduling instructions from control-equivalent basic blocks has been studied in [2]. A more general approach, based on integer linear programming and combining scheduling and global code motion, was proposed in [28]. Compared to local approaches, global schedulers use a larger scope to help them making decisions, and thus have potential to obtain a better schedule. Besides data dependences, global schedulers must also preserve control dependences.

6.2 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, 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 [14, 3] and clustered architectures [5, 18].

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

Motivated by CMPs, several multi-threaded scheduling techniques have recently been proposed to generate multi-threaded code from general-purpose, sequential applica-

Num. of Threads	Scope			
	Basic Block	Trace	Loop	Proc.
Single	List Sched. [17]	Trace [7, 5, 3] Superblock [11]	SWP [14, 18]	GSTIS [2] ILP [28]
Multiple	Space-time [15] Convergent [16] DAE Sched. [23]		DSWP [19]	*GMT*

Table 3. Instruction scheduling space.

tions [15, 16, 19]. Most of these techniques use a *local multi-threaded* (LMT) approach [15, 16]. LMT schedulers have to insert synchronization at branch instructions: before jumping to the next block, the thread taking the branch decision sends the branch direction to the other threads through the communication queues. The other threads then mimic this branch, so that all threads follow the same path through the program's control-flow graph (CFG) [15, 16]. A similar approach is used by schedulers for decoupled access/execute architectures, which may even use specialized queues to communicate branch directions [23].

Although suitable for single-threaded architectures, the problem of using local scheduling for multi-threaded machines is that it effectively only exploits *instruction-level parallelism* inside basic blocks. Unfortunately, general-purpose applications typically have a very small number of instructions per basic block, usually less than 10. Not surprisingly, existing compilers based on local scheduling for multi-threaded architectures have shown little effectiveness in extracting parallelism from general-purpose, sequential applications [15, 16].

In [19], we recently proposed a global multi-threaded scheduling technique, called decoupled software pipelining (DSWP). This technique utilizes separate threads to execute different stages of a loop in a pipelined fashion. Although DSWP is classified as global scheduling, it is limited to loop regions. Besides that, the technique described in [19] imposes some extraneous loop dependences, which we later proved unnecessary.

In this paper, we presented general *global multi-threaded* scheduling algorithms, which can simultaneous schedule instructions in arbitrary code regions. Table 3 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.

7 Conclusion

The recent trend in the microprocessor industry to build chip multiprocessors (CMPs) has increased the interest in automatic thread extraction for the large base of non-scientific applications. Despite this interest and CMPs' ability to simultaneously execute different control paths, existing multi-threading techniques have mostly been restricted to local scheduling approaches. This paper introduced the concept of global multi-threaded (GMT) instruction scheduling, a general technique that can exploit fine-grained thread-level parallelism on modern CMPs. Opposed to local approaches, GMT scheduling exposes thread-level parallelism, enabling the concurrent execution of instructions from different regions of the control-flow graph. This paper also described algorithms to find profitable GMT schedules for arbitrary code regions using a PDG representation, as well as an algorithm to generate multi-threaded code for any GMT schedule decision. Experimental results on a number of benchmarks demonstrated the enormous potential of our techniques.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[2] 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.

[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] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

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

[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] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[14] 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.

[15] 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 *The Proceedings of the Eighth International Confrence on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[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] 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.

[27] 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.

[28] 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.

[29] 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.