Communication Optimizations for Global Multi-Threaded Instruction Scheduling

Guilherme Ottoni David I. August

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

Abstract

The recent shift in the industry towards chip multiprocessor (CMP) designs has brought the need for multi-threaded applications to mainstream computing. As observed in several limit studies, most of the parallelization opportunities require looking for parallelism beyond local regions of code. To exploit these opportunities, especially for sequential applications, researchers have recently proposed *global multi-threaded instruction scheduling* techniques, including DSWP [16] and GREMIO [15]. These techniques simultaneously schedule instructions from large regions of code, such as arbitrary loop nests or whole procedures, and have been shown to be effective at extracting threads for many applications. A key enabler of these global instruction scheduling techniques is the *Multi-Threaded Code Generation* (MTCG) algorithm proposed in [16], which generates multi-threaded code for *any* partition of the instructions into threads. This algorithm inserts communication and synchronization instructions in order to satisfy all inter-thread dependences.

In this paper, we present a general compiler framework, COCO, to optimize the communication and synchronization instructions inserted by the MTCG algorithm. This framework, based on thread-aware data-flow analyses and graph min-cut algorithms, appropriately models and optimizes all kinds of inter-thread dependences, including register, memory, and control dependences. Our experiments, using a fully automatic compiler implementation of these techniques, demonstrate significant reductions (about 30% on average) in the number of dynamic communication instructions in code parallelized with DSWP and GREMIO. This reduction in communication translates to performance gains of up to 40%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — code generation, compilers, optimization; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors) — multiple-instruction-stream, multiple-data-stream processors (MIMD); G.2.2 [*Discrete Mathematics*]: Graph Theory — graph algorithms

General Terms Algorithms, Languages, Performance

Keywords multi-threading, instruction scheduling, communication, synchronization, data-flow analysis, graph min-cut

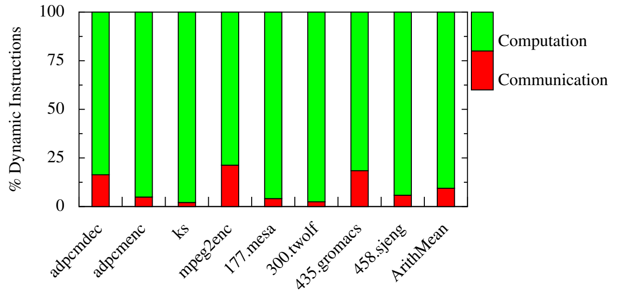
1. Introduction

The recent shift in the industry towards chip multiprocessors (CMP) has brought to mainstream computing the need to exploit thread-level parallelism (TLP) as a means to improve performance. Since developing parallel applications has long been recognized as significantly harder than developing sequential ones, having automatic tools to extract TLP from sequential programs is very attractive. Unfortunately, despite decades of research on parallelizing compilers, these have only proved effective in the restricted domain of scientific and data-parallel applications, which often have regular array-based memory accesses and little control flow.

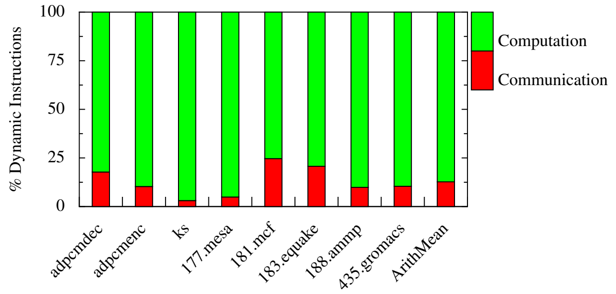
The irregular memory accesses and control flow in general-purpose applications typically result in a large number of dependences that have to be respected by the compiler. This makes it hard to find coarse-grained parallelism in these programs. In order to automatically extract TLP for these applications, researchers have investigated two main directions. First, thread-level speculation (TLS) techniques have been proposed to exploit optimistic parallelization in the presence of infrequent dependences [23, 30, 31]. Though effective in some cases, TLS generally requires expensive hardware support. Second, researchers have proposed hardware mechanisms to lower the inter-thread communication costs, thus enabling the exploitation of finer-grained, non-speculative TLP found in general-purpose applications [18, 19, 24]. Effectively exploiting this second kind of hardware mechanism to automatically extract fine-grained, *non-speculative* TLP is the topic of this paper.

Proposed hardware support mechanisms for non-speculative, fine-grained TLP 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 inserting *communication* instructions to satisfy inter-thread dependences using the hardware support. These hardware mechanisms enable TLP at a fine granularity, which is unsuitable for programmers to manually exploit. Therefore, generating code that exploits these TLP opportunities is better performed by a compiler's instruction scheduler.

In order to effectively extract TLP, several limit studies have shown that it is necessary to exploit parallelism beyond local regions of code and to execute multiple flows of control in parallel [10, 26]. In fact, CMPs' ability to simultaneously execute different regions of code on different cores is their key advantage over single-core processors. To effectively exploit these opportunities, researchers have recently proposed *Global Multi-Threaded* (GMT) instruction scheduling techniques, which simultaneously sched-



(a) GREMIO



(b) DSWP

Figure 1. Breakdown of dynamic instructions in code generated by (a) GREMIO and (b) DSWP.

ule instructions of a global region of code into multiple threads. Examples of such scheduling techniques are Decoupled Software Pipelining (DSWP) [16] and GREMIO [15], which are able to extract non-speculative TLP from arbitrary loop nests and procedures. Given that threads can pursue different paths of execution, a major difficulty faced by GMT scheduling techniques is to generate correct code while not inserting too much communication, which can negate the parallelism.

A key enabler of both DSWP and GREMIO, and virtually any GMT scheduling technique, is the very general *Multi-Threaded Code Generation* (MTCG) algorithm proposed in [16]. This algorithm takes any partition of instructions into threads and generates provably correct code. To achieve this, the MTCG algorithm inserts communication instructions to satisfy all inter-thread dependences. For GMT scheduling, these dependences include data dependences through registers and memory, as well as control dependences.

Because of the number of dependences in general-purpose applications, the overhead of communication instructions can be quite significant in code generated by GMT instruction scheduling techniques. For example, in Figure 1, we show the dynamic percentages of *communication* instructions compared to the original instructions in the program (the *computation*) for various benchmarks parallelized with GREMIO and DSWP. As illustrated, the communication instructions can account for up to one fourth of the total instructions. This motivates the study of communication optimizations. Reducing the number of dynamic communication instructions between threads not only improves the performance by reducing the number of instructions executed, but also reduces the contention and power consumption by the on-chip inter-connect.

In this paper, we present a general COmpiler Communication Optimization (COCO) framework to optimize inter-thread communication for the MTCG algorithm. COCO uses a set of novel *thread-aware* data-flow analyses, in combination with efficient graph min-cut algorithms. Based on these, we describe how to precisely model and optimize all kinds of inter-thread dependences, including register, memory, and control dependences. Our experiments demonstrate the effectiveness of these optimizations on a variety of benchmarks parallelized using DSWP and GREMIO.

2. GMT Instruction Scheduling

This section gives an overview of GMT instruction scheduling by describing the key components in virtually any such technique. In essence, GMT instruction scheduling encompasses three main steps, illustrated in Figure 2. The first step is to build a *Program Dependence Graph* (PDG) [5], including all the dependences that need to be respected. The PDG for an arbitrary global (intra-procedural) region must include both data and control dependences. In a low-level representation, data dependences can be through either registers or memory. Register dependences can be computed through simple data-flow analysis, while memory dependences require more complex alias analysis, typically based on pointer analysis. PDGs provide a good abstraction for performing global instruction scheduling because they contain all the dependences that need to be honored in order to preserve the semantics of the original program [21]. This implies that, whenever two dependent instructions are scheduled on different threads, some form of communication or synchronization has to be inserted in the code so that this dependence is respected.

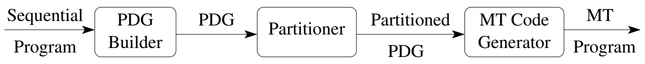


Figure 2. GMT instruction scheduling block diagram.

After the PDG is constructed, a GMT scheduler needs to assign instructions to threads, i.e. to *partition* the instructions among threads. This phase, the partitioner, is where the GMT scheduling techniques differ. For example, DSWP is a partitioner that creates a pipeline of threads, among which the dependences only flow in one direction [16]. The GREMIO technique [15], on the other hand, allows cyclic inter-thread dependences and schedules instructions based on their control relations and an estimate of when instructions will be ready to execute.

Once a partition of instructions into threads is chosen, the corresponding multi-threaded code needs to be generated. For this purpose, both DSWP and GREMIO utilize the general MTCG algorithm proposed in [16]. This algorithm is also responsible for inserting communication and synchronization in the multi-threaded code so as to satisfy all PDG's inter-thread dependences. Since this paper focuses on optimizing the communications generated by MTCG, we describe it in detail in Section 2.1. Notice that, as illustrated in Figure 2, the PDG representation and the MTCG algorithm together provide a nice framework to perform GMT instruction scheduling. Different GMT schedulers can be implemented simply by "plugging" different partitioners in this framework.

2.1 MTCG Algorithm

This subsection describes the MTCG algorithm [16], highlighting how it enforces inter-thread dependences. Algorithm 1 presents the pseudo-code for MTCG, which takes as input the original control-flow graph (*CFG*), the program dependence graph (*PDG*), and the chosen partition (*P*) of the instructions into threads. As output, this algorithm produces a new CFG for each of the resulting threads, containing its corresponding instructions and the necessary communication instructions.

Algorithm 1 MTCG

```
Require: CFG, PDG, $P = \{P_1, \ldots, P_n\}$
 1: for each thread $P_i$, create $CFG_i$ with its relevant basic blocks;
 2: insert each instruction in its thread's CFG;
 3: for all arcs $(I \rightarrow J) \in PDG$ do                    // insert communication
 4:     let $P_i, P_j$ be such that $I \in P_i$ and $J \in P_j$
 5:     if $P_i = P_j$ then
 6:         continue
 7:     end if
 8:     $q \leftarrow get\_free\_queue()$
 9:     if $dep\_type(I \rightarrow J) = Register$ then                // register dep.
10:         $r_k \leftarrow dependence\_register(I \rightarrow J)$
11:         $add\_after(CFG_i, I,$ "produce [q] = $r_k$")
12:         $add\_after(CFG_j, I,$ "consume $r_k$ = [q]")
13:     else if $dep\_type(I \rightarrow J) = Memory$ then             // memory dep.
14:         $add\_after(CFG_i, I,$ "produce.sync [q]")
15:         $add\_after(CFG_j, I,$ "consume.sync [q]")
16:     else                                                          // control dep.
17:         $r_k \leftarrow register\_operand(I)$
18:         $add\_before(CFG_i, I,$ "produce [q] = $r_k$")
19:         $add\_before(CFG_j, I,$ "consume $r_k$ = [q]")
20:         $add\_after(CFG_j, I, duplicate(I))$
21:     end if
22: end for
23: fix branch/jump targets.
```

In essence, the MTCG algorithm has four main steps. In the first step (line 1), a new CFG_i is created for each thread P_i. CFG_i contains one basic block for each *relevant basic block* to P_i in the original *CFG*. A basic block is *relevant* to a thread if it 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 where communication instructions are inserted by MTCG, as described shortly.

The second step of MTCG (line 2) 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.

The third step of the algorithm is to insert the inter-thread communication instructions, and is detailed in lines 3-22 in Algorithm 1. For each such dependence, a separate communication queue is used.[1] In order to preserve the conditions under which each dependence occurs, MTCG adopts this strategy: *each dependence is communicated at the point of its source instruction.* The actual communication instructions inserted depend on the type of the dependence, as illustrated in Algorithm 1. In this pseudocode, $add\_before(CFG_i, I, instr)$ inserts $instr$ in CFG_i at the point right before instruction I's position in the original *CFG*, and add_after works analogously. Register dependences are implemented by communicating the register in question. 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 target thread, 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. A simple optimization (not illustrated in Algorithm 1) is that, if an instruction is the source of multiple dependences with the same target thread, it only needs to be communicated once.

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

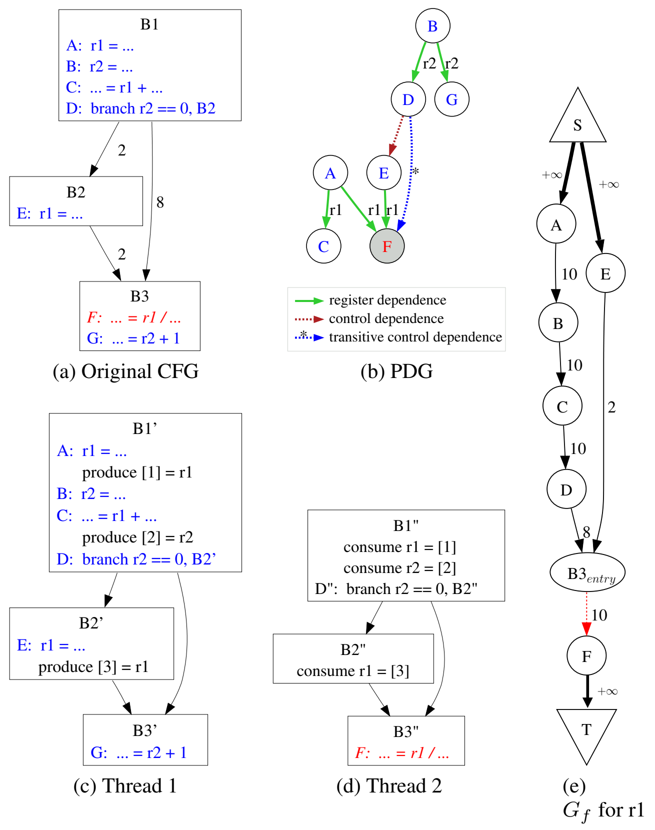


Figure 3. Simple example of the MTCG algorithm.

The last step of the MTCG algorithm (line 23) is to adjust the branch and jump targets in each new CFG. 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. Section 2.2.3 in [16] describes how to compute the correct branch targets, based on the post-dominance relation.

The simple example in Figure 3 illustrates how MTCG works. Figure 3(a) shows the original code in a CFG, and Figure 3(b) contains the corresponding PDG. Assume a partition into 2 threads: $P_1 = \{A, B, C, D, E, G\}$ and $P_2 = \{F\}$. Figures 3(c)-(d) show the code generated by MTCG for each thread. Thread 1 (P_1) has instructions in all basic blocks, and thus they are all relevant to it. For thread 2, B3 is relevant because it holds instruction F assigned to this thread, and B1 and B2 are relevant because they contain instructions on which F depends. As can be seen in the PDG, there are 3 inter-thread dependences in this case: two register dependences $(A \rightarrow F)$ and $(E \rightarrow F)$ involving $r1$, and a transitive control dependence $(D \rightarrow F)$. This transitive control dependence exists because D controls E, which is the source of an inter-thread register dependence. Transitive control dependences are necessary in order to implement the correct condition under which a dependence occurs [16]. In Figures 3(c)-(d), there is a pair of `produce` and `consume` instructions for each inter-thread dependence, inserted according to Algorithm 1.

In the example in Figure 3, the set of communication instructions inserted by MTCG is not optimal. It would be more efficient to simply communicate $r1$ at the beginning of blocks B3' and B3", for two reasons. First, this would avoid communicating $r1$ twice on the path (B1, B2, B3). Second, this would make it unnecessary to have branch D'' in thread 2, thus saving the communication of $r2$ as well. The next section describes efficient and effective algorithms to perform these register communication and control flow optimizations in general, as well as to optimize the memory synchronizations inserted by MTCG.

3. Communication Optimizations

As motivated by the example in Figure 3, the goal of COCO is to reduce the number of dynamic communication and synchronization instructions executed in the generated MT code. Unfortunately, as with many program analysis and optimization problems, this problem is undecidable in general, since it is undecidable to statically determine which paths will be dynamically executed. For this reason, COCO uses a profile-based approach, in which an estimate on the execution count of each CFG edge is available. These estimates can be obtained through profiling or through static analyses, which have been demonstrated to be also very accurate [28]. Given the profile weights, the problem can be formulated as to minimize the communication assuming each CFG edge's execution frequency as indicated by its weight.

Before formulating the communication optimization problems and describing the algorithms, we introduce several definitions and properties that are necessary. First, in Definition 1 below, we define the notion of *relevant branches* to a thread. This definition parallels the notion of relevant basic blocks of [16], described earlier in Section 2.1. However, Definition 1 is more general in that it allows communication to be inserted at points other than the point of a dependence's source instruction.

DEFINITION 1 (Relevant Branches). *A branch instruction B is* relevant *to thread T if either:*

1. *B is assigned to T in the partition; or*
2. *B controls the insertion point of an input dependence of an instruction assigned to T; or*
3. *B controls another branch B' relevant to T.*

The intuition is that relevant branches are those that thread T will contain, either because they were assigned to this thread or because they are needed to implement the correct condition under which a dependence into T must happen.

Based on the notion of relevant branches for a given thread, we define its *relevant points* in the program.

DEFINITION 2 (Relevant Points). *A program point p in the original CFG is* relevant *to thread T iff all branches on which p is control dependent are relevant branches to T.*

In other words, the relevant points for thread T are those that depend only on T's relevant branches. This means that the condition of execution of these points can be implemented without adding new branches to T.

The communication instructions generated by MTCG obey an important property to enable the TLP intended by the thread partitioner:

PROPERTY 1. *All inter-thread communications in the generated MT code correspond to dependence arcs in the PDG, including transitive control dependence arcs.*

This property guarantees that only dependences represented in the PDG will be communicated. This is important because, as illustrated in Figure 2, the partitioner is based on the PDG. If Property 1 was not respected, then MTCG could hurt the parallelism intended by the partitioner. For example, the DSWP partitioner assigns instructions to threads so as to form a pipeline of threads. However, if Property 1 is not respected, a dependence cycle among the threads can be created.

In the MTCG algorithm, since communication is always inserted at the point corresponding to the source of the dependence, *all* inter-thread transitive control dependences need to be implemented [16]. In other words, each of these dependences will require its branch operands to be communicated and the branch to be duplicated in the target thread. With COCO, however, a better placement of data communications can reduce the transitive control dependences that need to be implemented. For this reason, besides Property 1, COCO also respects the following property to limit the transitive control dependences that need to be implemented:

PROPERTY 2. *The communication instructions to satisfy a dependence from thread T_s to thread T_t must be inserted at relevant points to T_s.*

Essentially, this property prevents branches from becoming relevant to thread T_s merely for implementing a dependence emanating from T_s. In other words, no new branches must be added to T_s in order to implement a dependence from it to T_t. However, additional branches may be made relevant to the target thread T_t in order to implement one of its input dependences.

Besides Properties 1 and 2, the placement of register communications must also respect the *Safety* property for correctness:

PROPERTY 3 (Safety). *A register dependence from T_s to T_t involving (virtual) register r must be communicated at* safe *points, where T_s has the* latest *value of r.*

This property is necessary for correctness because communicating r at an unsafe point would, in some control paths, overwrite r in T_t with a stale value. In other words, a dependence from a definition of r not in T_s to a use of r in T_t would not be respected, thus changing the program's semantics. Notice that MTCG's placement of communication is safe, since the registers are communicated immediately after they are defined.

The set of registers that are safe to communicate from thread T_s to any other thread at each program point can be precisely computed using the data-flow equations (1) and (2) below. In these equations, DEF_{T_s} and USE_{T_s} denote the set of registers defined and used by instruction n if it is assigned to T_s, and DEF means the set of registers defined by n regardless of which thread contains n.

$$\begin{aligned}\text{SAFE}_{\textit{out}}(n) &= \text{DEF}_{T_s}(n) \cup \text{USE}_{T_s}(n) \cup \\ &\quad (\text{SAFE}_{\textit{in}}(n) - \text{DEF}(n)) \qquad (1)\\ \text{SAFE}_{\textit{in}}(n) &= \bigcap_{p \in \text{Pred}(n)} \text{SAFE}_{\textit{out}}(p) \qquad (2)\end{aligned}$$

The intuition behind these equations is that T_s is guaranteed to have the latest value of a register r right after T_s either defines or uses r. Furthermore, the value of r in T_s becomes stale after another thread defines r. Finally, the data-flow analysis to compute safety is a forward analysis, and the SAFE_{in} sets are initially empty.

This safety data-flow analysis is said to be *thread-aware* because, although operating on a single CFG, it takes the assignment of instructions to threads into account. An equivalent analysis could operate on multiple CFGs (one per thread) simultaneously, given the correspondence between the basic blocks in all CFGs.

Based on the definitions and properties above, the next subsection focuses on optimizing the communication between a pair of threads. Later, in Section 3.2, a general algorithm to handle any number of threads is described.

3.1 Optimizing a Pair of Threads

We first describe how COCO optimizes register communication in Section 3.1.1. Then, Section 3.1.2 shows how to extend COCO to also minimize control flow. Finally, Section 3.1.3 demonstrates how COCO optimizes memory synchronizations as well.

3.1.1 Optimizing Register Communications

We now formulate the problem of optimizing register communications from a source thread T_s to a target thread T_t. Since the communication of each register requires its own set of instructions, it is possible to optimize the communication of each register independently. So let r denote the register whose communication is to be optimized.

The register communication optimization problem can be precisely modeled as a *min-cut problem in directed graphs*, by constructing a graph $G_f = (V_f, A_f)$ derived from the CFG as described shortly. The intuition behind the construction of G_f is that a cut in this graph will correspond to communicating r at the program points corresponding to the arcs in this cut.

The set of vertices in V_f contains the original code instructions where r is *live with respect to* T_t. That means the live range of r considering only the uses of r in the instructions assigned to T_t. This can be computed using a thread-aware data-flow analysis very similar to the standard liveness analysis.[2] In addition, V_f also contains one vertex corresponding to the entry of each basic block where r is live with respect to T_t. The need for these vertices will become clear shortly. Finally, there are two special nodes: a source node S, and a target (or sink) node T. The arcs in A_f are of two kinds. *Normal arcs* represent possible flows of control in the program, corresponding to the arcs in the CFG constructed at the granularity of instructions. These arcs have a *cost* equal to the profile weight of their corresponding CFG arcs. In addition, there are also *special arcs* from S to every definition of r in T_s, and from every use of r in T_t to T. The costs of the special arcs are set to infinity to prevent them from participating in a minimum cut. This is necessary because special arcs do not correspond to program points, and thus cannot have communication instructions placed on them.

As an example, consider the code in Figure 3(a) with the partition $T_s = \{A, B, C, D, E, G\}$ and $T_t = \{F\}$. Figure 3(e) illustrates the graph G_f for register $r1$. The source and sink nodes are represented by a triangle and an inverted triangle, respectively. Node $B3_{entry}$ corresponds to the beginning of block B3, the only block that has $r1$ live at its entry.

When drawing special arcs to the target node T in G_f, besides the uses of r in instructions assigned to T_t, uses of r in relevant branches to T_t are also considered as uses in T_t. The reason for this is that, as mentioned earlier, relevant branches to a thread need to be included in it to properly implement its control flow. In effect, treating branches as belonging to all threads to which they are relevant allows the communication of branches' register operands to be optimized along with register data communications. This can result in better communication of branch operands compared to MTCG's strategy of always communicating branch operands immediately before branches in order to implement control dependences (lines 18-19 in Algorithm 1).

Notice that, in order to satisfy the dependences involving r from T_s to T_t, communication instructions for r can be inserted at any subset of A_f that disconnects T from S. In other words, any *cut* in G_f corresponds to a valid placement of communication instructions, namely communicating r at each arc in this cut. This guarantees that r will be communicated from T_s to T_t along every path from a definition of r in T_s to a use of r in T_t. In particular,

[2] The data-flow equations are omitted here in interest of space.

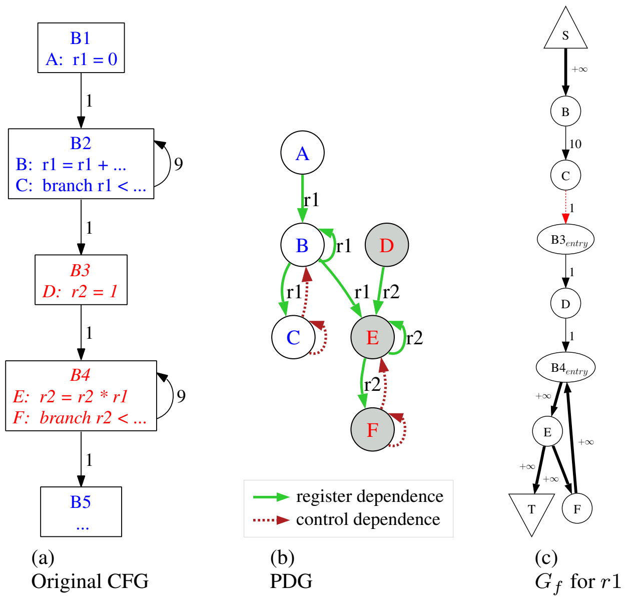


Figure 4. An example with loops.

the original MTCG algorithm always uses the cut containing the outgoing arcs in G_f of the instructions defining r in T_s. This corresponds to the cut containing $(A \rightarrow B)$ and $(E \rightarrow B3_{entry})$ in Figure 3(e). Since S only has arcs to nodes corresponding to T_s's instructions defining r, this is clearly a cut in G_f.

By the construction of G_f, for a given cut C in G_f, the number of dynamic communications of r that will be executed corresponds to the cost of the arcs in C. Therefore, the problem of finding the minimum number of dynamic communications reduces to finding a *minimum cut* in G_f. In Figure 3(e), arc $(B3_{entry} \rightarrow F)$ alone forms a min-cut, with a cost of 10. This example also illustrates the role of the nodes in G_f corresponding to basic block entries, which is to allow the placement of communications before the first instruction in a basic block (B3 in this case).

In fact, the formulation described so far is still incomplete, because it allows any normal arc in G_f to be cut. However, there are arcs that must not participate in a cut because communicating r at those points violates one of Properties 1, 2 or 3. To prevent such arcs from participating in a cut, their costs are set to infinity. As long as there exists a cut with finite cost, these arcs (and also the special arcs involving S and T) are guaranteed not to be in a min-cut. Fortunately, a finite-cost cut always exists: the cut that the MTCG algorithm picks. That is true because the points right after the definitions of r in T_s are both safe (i.e. T_s has the latest value of r there) and relevant to T_s (since they have the same condition of execution of the definitions of r in T_s).

To illustrate a more drastic case where the MTCG algorithm generates inefficient code, consider the example in Figure 4. The partition in this case is $T_s = \{A, B, C\}$ and $T_t = \{D, E, F\}$. The only inter-thread dependence is the register dependence $(B \rightarrow E)$. The MTCG algorithm communicates $r1$ right after instruction B, inside the first loop. For this reason, a transitive control dependence $(C \rightarrow E)$, not illustrated in Figure 4(b), also needs to be communicated. As a result, thread T_t will contain the first loop as well. In effect, T_t will consume the value of $r1$ each time B is executed, even though only the last value assigned to $r1$ is used by instruction E. Figure 4(c) shows the graph G_f constructed for $r1$. Notice that G_f does not contain nodes before B, including the arc

$(C \rightarrow B2_{entry})$, since $r1$ is not live with respect to T_t at these points. Applying the register communication optimization, $r1$ can be communicated in either of the arcs with cost 1 in Figure 4(c). Any of these cuts essentially corresponds to communicating $r1$ at block B3. This drastically reduces the number of times $r1$ is communicated from the total number of B2's loop iterations, 10, down to 1. Furthermore, as a side effect, this completely removes the first loop from thread T_t, making it unnecessary to implement the transitive control dependence $(C \rightarrow E)$.

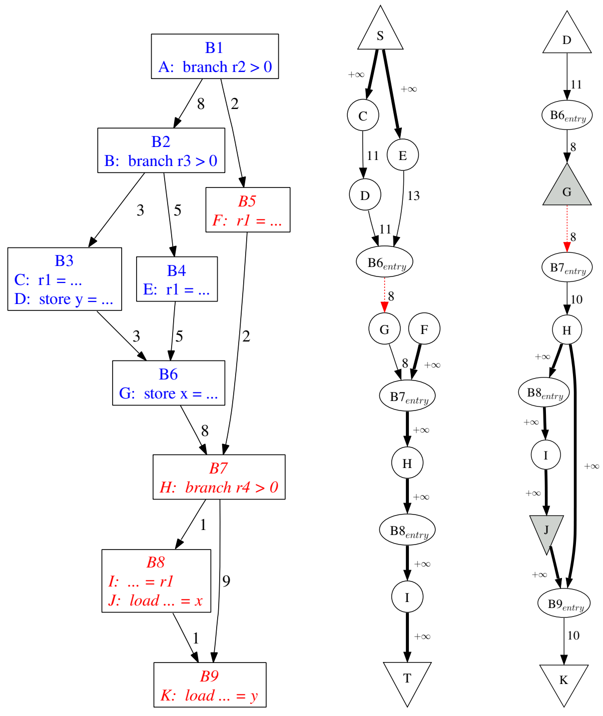
Fortunately, there are efficient algorithms to compute a min-cut in direct graphs. In fact, due to its duality to maximum flow [6], min-cut can be solved by efficient and practical max-flow algorithms based on preflow-push, with worst-case time complexity $O(n^3)$, where n is the number of vertices [4]. For our problem, given that G_f is limited to a register's live-range, even algorithms with worse time complexity run fast enough so as to not increase compilation time significantly, as observed in our experiments.

3.1.2 Reducing Control Flow

As illustrated in the example from Figure 4, the placement of data communication can also reduce the control flow in the target thread. In some cases, as in Figure 4, this comes for free simply by optimizing the data communications. However, there are cases where there are multiple cuts with the minimum cost, but some of them require more inter-thread control dependences to be implemented than others. Extra control flow in the target thread T_t is necessary whenever communication is placed at points currently not relevant to T_t. This forces these branches to be added to the set of relevant branches for T_t, so that they will need to be implemented in T_t.

In order to avoid branches unnecessarily becoming relevant to T_t, the costs of the arcs in G_f can be adjusted as follows. The idea is to penalize arcs that, if cut, will require additional branches to become relevant to T_t. Thus, we add to each arc A in G_f the profile weight of each currently irrelevant branch to T_t that will become relevant if communication is placed on A. The reasoning is that these branches would not be necessary otherwise, so we add the number of dynamic branches that would be executed to the cost of A. To illustrate this, consider the example in Figure 5(a), with $T_s = \{A, B, C, D, E, G\}$ and $T_t = \{F, H, I, J, K\}$. The corresponding PDG is shown in Figure 5(d). Consider the communication of $r1$ from T_s to T_t. This communication is only allowed to be placed in basic blocks B3, B4, and B6, since, from B7 on, it is not safe due to the definition of $r1$ in instruction F in T_t. The two alternatives then are to communicate $r1$ either in B6 or in B3 and B4. Looking at the profile weights, both alternatives look equally good. However, communicating at B3 and B4 makes the branch instruction B relevant to T_t, while communicating at B6 does not. Figure 5(b) illustrates the graph G_f for $r1$ with the costs adjusted to account for control flow costs. Assume branch B is currently not relevant to T_t. The arcs $(C \rightarrow D)$, $(D \rightarrow B6_{entry})$, and $(E \rightarrow B6_{entry})$ are control dependent on B, and thus have the profile weight of B, 8, added to their costs. With these penalties added, the min-cut in G_f is either arc $(B6_{entry} \rightarrow G)$ or arc $(G \rightarrow B7_{entry})$ in Figure 5(b). Both these cuts correspond to placing the communication of $r1$ in block B6, thus avoiding adding branch B to T_t's set of relevant branches.

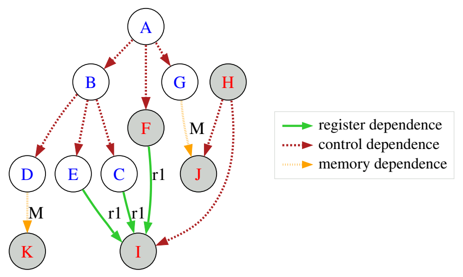
Notice that, after adding these penalties to account for control flow, the problem is not precisely modeled anymore. For instance, multiple arcs including a penalty for one branch will include the cost of this branch, and thus a cut including two or more of these arcs will be over-penalized. But, since the arc costs are used to choose a cut, the information about which arcs will participate in the solution cut is unknown a priori to make the control-flow penalties more precise.



(a)
Original CFG

(b)
G_f
for $r1$

(c)
G_f for
mem.



(d) PDG

Figure 5. An example including memory dependences.

3.1.3 Optimizing Memory Synchronizations

This section describes how synchronization instructions, used to respect inter-thread memory dependences, can also be accurately modeled in terms of graph min-cut.

Although memory dependences are also implemented through queues and `produce` and `consume` instructions (Section 2.1), they differ from register dependences in several aspects. First, for memory dependences, no actual operand is sent through the queues, and only the synchronization matters. As a result, multiple memory dependence arcs involving unrelated memory locations can share the same synchronization instructions. This changes a fundamental characteristic of the optimization problem, as described shortly. Another difference compared to register communication is that, for memory, the `produce` and `consume` instructions must have the proper *release* and *acquire* semantics in the memory subsystem. That is, the `produce` must ensure that previous memory-related instructions have committed and, analogously, the `consume` must commit before successive memory-related instructions execute. Compared to the `produce` and `consume` instructions for register communication, the memory versions restrict reordering of instructions in the microarchitecture. This difference is also the reason why register communication instructions cannot be used to satisfy memory dependences.

As mentioned above, the fact that memory dependence arcs involving disjoint sets of memory locations can share synchronization instructions makes the problem different from the register communication one. The reason for this derives from the fact that, while the single-source, single-sink min-cut problem can be efficiently solved in polynomial time, min-cut is NP-hard for multiple source-sink pairs (also called *commodities*), where the goal is to disconnect each source from its corresponding sink [7]. For registers, the problem could be precisely modeled using a single source-sink pair by applying the standard trick of creating special source and sink nodes and connecting them to the rest of the graph appropriately. This was possible because, for a register r, it was necessary to disconnect *every* definition of r in T_s from *every* use in T_t. For memory, however, a similar trick does not lead to an optimal solution. Finding the optimal solution for memory requires optimizing all memory dependences from T_s to T_t simultaneously, since they all can be implemented by the same synchronization instructions. Nevertheless, it is *not necessary* for all sources of memory dependences in T_s to be disconnected from all targets of these dependences in T_t, since dependences can refer to disjoint sets of memory locations. A memory instruction in T_s needs only to be disconnected from its dependent memory instructions in T_t. To accurately model this optimization problem, it is necessary to use a graph min-cut with multiple source-sink pairs.

Notice that, although this difference requires using sub-optimal algorithms for memory optimization in practice, the possibility of sharing synchronization instructions makes the potential optimization impact larger for memory than for registers. In fact, this is confirmed in the experiments in Section 4.

We now describe how to construct the graph G_f to optimize memory dependences from thread T_s to thread T_t. Although the register optimization could be restricted to the region corresponding to the register's live-range, this is not always possible for memory due to weak memory updates and the impossibility of eliminating all false dependences through renaming. For this reason, the nodes in G_f for memory may need to correspond to the entire region being parallelized. Akin to what was described for registers, G_f here includes nodes corresponding to basic block entries. As explained above, to precisely model the memory optimization problem, it is necessary to use multiple source-sink pairs in G_f. Specifically, for each memory dependence arc from T_s to T_t in the PDG, a source-sink pair is created for its source and target instructions.

The costs on G_f's arcs for memory are the same as for registers, with two differences. First, there is no notion of safety for memory synchronization. In other words, since no operand is communicated for memory dependences, no arc is prohibited from participating in the cut because it is not safe to synchronize at that point. The second difference is that, since the source (sink) nodes here correspond to real instructions in the program, their outgoing (incoming) arcs are allowed to participate in the cut and thus do not have their costs set to infinity. Similar to what is used for register dependences, arcs dependent on irrelevant branches to either T_s or T_t have their costs set specially.

As an example, consider the placement of memory synchronization for the code in Figure 5. There are two cross-thread memory dependences from T_s to T_t: $(D \rightarrow K)$ involving variable y, and $(G \rightarrow J)$ involving variable x. Figure 5(c) illustrates the G_f graph constructed as described, with the two source-sink pairs distinguished by different shades. The arcs from node H all the way down to $B9_{entry}$ have infinite cost because they are control dependent on branch H, which is not relevant to T_s. The min-cut solution in the example is to cut the arc $(G \rightarrow B7_{entry})$, with a cost of 8.

Algorithm 2 COCO

```
Require: CFG, PDG, P
 1: $G_T \leftarrow$ Build_Thread_Graph($PDG$, $P$)
 2: $relevantBr \leftarrow$ Init_Relevant_Branches($G_T$, $P$)
 3: $deps \leftarrow \emptyset$
 4: repeat
 5:   $oldDeps \leftarrow deps$
 6:   $deps \leftarrow \emptyset$
 7:   for all arcs $(T_s \rightarrow T_t) \in G_T$ [in topological order] do
 8:     $stDeps \leftarrow \emptyset$
 9:     for all registers $r$ to be communicated from $T_s$ to $T_t$ do
10:       $G_f \leftarrow$ Build_Flow_Graph_Register($r, T_s, T_t, relevantBr$)
11:       $commArcs \leftarrow$ Min_Cut($G_f$)
12:       $stDeps \leftarrow stDeps \cup \{(r, T_s, T_t, commArcs)\}$
13:     end for
14:     $G_f \leftarrow$ Build_Flow_Graph_Memory($T_s, T_t, relevantBr$)
15:     $commArcs \leftarrow$ Min_MultiCut($G_f$)
16:     $stDeps \leftarrow stDeps \cup \{(MEM, T_s, T_t, commArcs)\}$
17:     Update_Relevant_Branches($relevantBr[T_t]$, $stDeps$)
18:     $deps \leftarrow deps \cup stDeps$
19:   end for
20: until $oldDeps = deps$
21: return $deps$
```

Given the NP-hardness of the min-cut problem with multiple source-sink pairs, the following heuristic solution is used in this work. The optimal single-source-sink algorithm is successively applied to each source-sink pair. When an arc is cut to disconnect a pair, it is removed from the graph so that this can help disconnecting subsequent pairs. As illustrated in our experiments, this simple heuristic performs well in practice.

3.2 Optimizing Multiple Threads

We now turn to optimizing the communication for multiple threads. COCO tackles this more general problem by relying on the pairwise algorithms described above.

Algorithm 2 presents the pseudo-code for COCO. As input, COCO takes the original CFG and PDG for the region being parallelized, as well as the partition into threads specified by the partitioner. As output, this algorithm returns the set of inter-thread dependences annotated with the points in the program where the communication instructions should be inserted. These annotations can be directly used to place communications in a slightly modified version of MTCG.

The first step in COCO (line 1) is to build a *thread graph* G_T as follows, representing the dependences between threads. For each thread, there is a node in G_T. There is an arc $(T_s \rightarrow T_t)$ in G_T if and only if there is a PDG dependence arc from an instruction in thread T_s to an instruction in another thread T_t. COCO successively optimizes the communications between each pair of threads connected by an arc in G_T.

The algorithm iteratively computes a set of inter-thread dependences (*deps*) annotated with their corresponding communication insertion points. Besides that, the algorithm maintains the set of relevant branches to each thread, computed according to Definition 1. At the beginning (line 2 in the algorithm), the sets of relevant branches are initialized following rules 1 and 3 in Definition 1. Later, as the insertion points for communication are computed, these sets grow using rules 2 and 3 in this definition. Although not illustrated in Algorithm 2, the set of relevant points to each thread, derived from the set of relevant branches according to Definition 2, is also maintained.

The algorithm iterates until the set of dependences with insertion points converges (*repeat-until* in lines 4-20). Iteration is neces-

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

(a) 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
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

(b) Selected benchmark functions.

Figure 6. Experimental setup.

sary in general because, to satisfy the input dependences of a thread T_i, other branches may become relevant to it. However, changing the set of T_i's relevant branches can affect the best placement for T_i's output dependences. That is because, to satisfy Property 2, no communication is allowed on irrelevant points for the source thread. Iteration can, however, be avoided in the special case when G_T is acyclic, by computing the placement for a thread's input dependences before for its output dependences.

The *for* loop in lines 7-19 computes the placement of communication for each pair of threads connected by an arc in the thread graph. As mentioned above, following a (quasi-)topological order of G_T's arcs here reduces the number of iterations of the *repeat-until* loop. For each arc $(T_s \rightarrow T_t)$ in G_T, the placement of communication is computed as described in Sections 3.1.1 through 3.1.3 above. That is, each register is optimized separately, and all memory dependences are optimized simultaneously. In each case, optimizing the communication placement involves creating a flow graph with costs on arcs, and then computing a min-cut in this graph. A tuple indicating the register involved in the dependence (or memory), its source and target threads, along with the communication insertion points computed, is then inserted in the set of dependences. Finally, on line 17, the set of relevant branches for the target thread is augmented to account for new branches that just became relevant to satisfy some dependences.

Algorithm 2 is guaranteed to converge because the sets of relevant branches are only allowed to grow, and the number of branches in the region is obviously finite.

Similar to code generated by the original MTCG algorithm, the code produced using COCO is also guaranteed to be deadlock-free. In both cases, this can be proved using the fact that pairs of `produce` and `consume` instructions are inserted at corresponding points in the original code.

4. Evaluation

In this section, we describe our experimental setup and results. Our experimental setup includes the compiler infrastructure in which COCO was implemented, the target architecture, and the benchmark programs used. Based on these, we then present results that demonstrate COCO's effectiveness in reducing the communication instructions and improving performance of the applications.

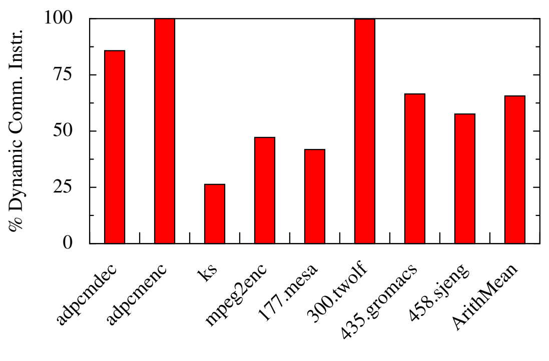
We implemented the COCO framework in the VELOCITY compiler [25], a multi-threading research compiler that targets Itanium 2. Two recently proposed global multi-threaded instruction scheduling techniques are implemented in VELOCITY: GREMIO and DSWP. To support these two techniques, VELOCITY uses the MTCG algorithm. VELOCITY uses the front-end of the IMPACT compiler [22] to obtain an assembly-level intermediate representation (IR). All traditional code optimizations are performed in VELOCITY, as well as some Itanium 2 specific optimizations. The global MT scheduling techniques in VELOCITY are 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 single-threaded instruction scheduling pass.

To evaluate the performance of the code generated by VELOCITY, we used a cycle-accurate CMP model comprising two Itanium 2 cores. The cores are validated, with IPC and constituent error components accurate to within 6% of real hardware for benchmarks measured [17]. In our model, the cores are connected by the *synchronization array* communication mechanism proposed in [19]. In Figure 6(a), we provide details about the simulator model.

The synchronization array (SA) in the model works as a set of low-latency queues. In our implementation, there are a total of 256 queues, each with a single element. For DSWP, which focuses on pipeline parallelism, queues with 32 elements are used. The SA has a 1-cycle access latency, and it 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 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.

For our experiments, we used the set of benchmarks that VELOCITY is currently able to parallelize using either DSWP or GREMIO, targeting two threads. These techniques were applied to the applications from the MediaBench, SPEC-CPU, and Pointer-Intensive benchmark suites that currently go through our toolchain. To reduce simulation time, the parallelization and simulations were restricted to important functions in these benchmarks, corresponding to at least 25% of the benchmark execution. In Figure 6(b), we list the selected application functions along with their corresponding benchmark execution percentages. VELOCITY has profile-weight information annotated on its IR, which was used for the costs on the G_f graphs for min-cut. The profiles were collected on smaller, *train* input sets, while the results presented here were run on larger *reference* inputs.

In Figure 7, we show the percentages of dynamic communication instructions that execute with COCO applied, relative to the codes using the original MTCG algorithm's communication strategy. The average reduction of the dynamic communication instructions was 34.4% for GREMIO, and 23.8% for DSWP. The largest


(a) GREMIO

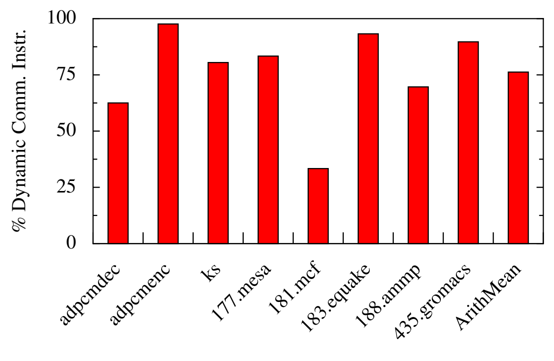
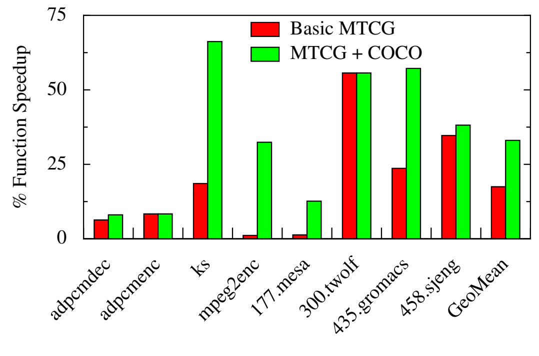

(b) DSWP

Figure 7. Relative dynamic communication / synchronization instructions after applying COCO.


(a) GREMIO

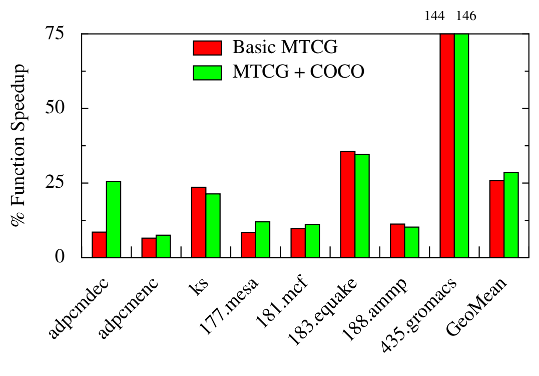

(b) DSWP

Figure 8. Speedup over single-threaded execution, without and with COCO.

reduction occurred for *ks* with GREMIO (73.7%), where an inner-loop whose only purpose was to consume a live-out could be completely removed from a thread, similar to the example in Figure 4. In only one smaller case, *adpcmenc* with GREMIO, COCO had no opportunity to reduce communication. COCO never resulted in an increase in dynamic communication instructions. Only two applications, *177.mesa* and *435.gromacs* with GREMIO, had inter-thread memory dependences. For both of these, COCO was able to remove more than 99% of the dynamic memory synchronizations.

COCO had a smaller impact on codes partitioned by DSWP in part because no inter-thread memory dependences can happen in this case. That is because the PDG employed in this work uses the results of a points-to static analysis [14]. And, since the instructions are inside a loop, any memory dependence is essentially bi-directional, thus forcing these instructions to be assigned to the same thread in order to form a pipeline [16]. With more powerful, loop-aware memory disambiguation techniques to eliminate false memory dependences, such as shape analysis or array-dependence analysis, DSWP could benefit more from COCO.

In Figure 8, we present the speedups for the benchmarks parallelized with GREMIO and DSWP over their single-threaded versions. For each benchmark, two bars are illustrated: the first using the basic MTCG algorithm, and the second using MTCG with COCO. In general, the speedups correlate with the reduction of dynamic communication instructions shown in Figure 7. The average speedup for GREMIO improves by 15.6%, while the average improvement is 2.7% for DSWP. The maximum speedup is for *ks* with GREMIO, for which COCO provided an extra 47.6% speedup. For *mpeg2enc*, COCO optimized the register communication in various hammocks, also significantly reducing the control flow in the generated threads. For the *435.gromacs* benchmark, DSWP resulted in a 2.44× speedup with MTCG, and 2.46× with MTCG+COCO. The high speedups for this benchmark resulted from an effective use of the doubled L2 cache capacity (the cores have private L2). In general, COCO improves performance not only by reducing the number of dynamically executed instructions, but also by increasing TLP through the removal of memory synchronizations and control dependences. For memory synchronizations, the reason is that the `consume.sync` instructions must wait for their corresponding synchronization token to arrive. For control dependences, the reason is that Itanium 2 uses a *stall-on-use* strategy, and control dependences are implemented as replicated branches that actually use their register operands. Removing register dependences has less effect because an outstanding `consume` instruction does not stall the pipeline until its consumed register is actually used.

For a couple of cases, COCO, although reducing the communication instructions, degraded performance slightly. The reason for this was a bad interaction with the later single-threaded instruction scheduler, which plays an important role for Itanium 2. Two possible solutions are being investigated to avoid this problem. One is to add, in the graph used for min-cut, penalties to arcs to take scheduling restrictions into account. Another alternative is to change the priority of the `produce` and `consume` instructions in the single-threaded scheduler.

Our current implementation of COCO uses Edmonds-Karp's min-cut algorithm [4], which has a worst-case time complexity of $O(n \times m^2)$, where n and m are the number of vertices and arcs in the graph, respectively. Since for CFGs m is usually $\Theta(n)$, this worst-case complexity approximates $O(n^3)$ in practice. In our experiments, this algorithm performed well enough not to significantly increase VELOCITY's compilation time. For production compilers, faster min-cut algorithms can be employed if necessary.

5. Related Work

This section compares the communication optimizations used by COCO to the most related work in the literature. Instruction scheduling for multi-threaded architecture is a relatively new research topic. Two existing global MT instruction scheduling techniques, DSWP [16] and GREMIO [15], are both based on the MTCG algorithm. In the previous sections, we have already showed the benefits of COCO for both these techniques. Although not evaluated in this work, COCO should also benefit a recently proposed technique that combines speculation with DSWP [27], which also uses MTCG.

Local MT instruction (LMT) scheduling techniques differ from the GMT in that they duplicate most of the program's CFG for each thread, thus mostly exploiting instruction-level parallelism within basic blocks. Similar to GMT, LMT techniques also need to insert communication instructions in order to satisfy inter-thread dependences. The Space-Time scheduling [11] LMT technique uses several simple invariants to make sure each thread gets the latest value of a variable before using it. First, each variable is assigned to a home node, which is intended to contain the latest value assigned to this variable. Second, each thread/node that writes to that variable communicates the new value to the home node right after the new value is computed. Finally, at the beginning of each basic block that uses a variable in a thread other than its home, a communication of this variable from its home node is inserted. This strategy is somewhat similar to the one used in the original MTCG algorithm, and other LMT techniques use similar approaches [12, 20]. Given their similarity to the original MTCG algorithm's strategy, they could also benefit from COCO to reduce the communication instructions inserted.

For clustered single-threaded architectures, the scheduler also needs to insert communication instructions to move values from one register bank to another [2, 13, 29]. However, the fact that dependent instructions are executed in different threads makes the generation and optimization of communication more challenging for multi-threaded architectures. The technique of [2] also uses graph partitioning algorithms.

Another piece of related work is the compiler communication optimization proposed for Thread-Level Speculation (TLS) [30]. There are several differences between the communication optimizations for TLS and GMT scheduling. First, each thread in TLS operates on a different loop-iteration, and therefore there are clear notions of order and of which thread has the latest value of a variable. Second, the communication between the threads is always uni-directional for TLS. Third, each thread only receives values from one upstream thread and sends values to one downstream thread. All these differences make the problem for TLS significantly simpler, so that algorithms based on partial redundancy elimination (PRE) [9] can be used to minimize the communication. For GMT scheduling, however, the problem is more complicated, and PRE does not solve it. For instance, in the register communication of $r1$ in Figure 5, there is no program point where $r1$ is *down-safe* (i.e., it will surely be used after that) and that satisfies our safety property (Property 3). The reason for this is that, depending on which path is executed, a different thread will have the latest value of $r1$.

Communication optimizations are also important for compiling data-parallel applications for distributed-memory machines [1, 3, 8]. The main differences from the problem there and the one studied in this paper are the following. First, there is an enormous discrepancy in the parallelism available in the applications, and how the parallelism is expressed by the programmer. This allows message-passing compilers to concentrate on a more regular style of parallelism, SPMD (single program multiple data), where all processors execute the same code. The irregular structure and fine granularity of the parallelism available in general-purpose applications require GMT scheduling to exploit more general forms of parallelism. Furthermore, the main communication optimization for message-passing systems is *communication combination*, where multiple messages are combined in a larger message to amortize overhead. Since GMT scheduling uses a scalar communication mechanism, these optimizations are not applicable in this context. In spirit, the optimizations proposed in this paper are closer to *redundancy optimizations* for distributed-memory systems. However, the techniques for data-parallel codes are very different, being strongly based on loops and array accesses and frequently unable to handle arbitrary control flow [8]. Another optimization proposed for message-passing systems is *pipelining*, where the message is sent earlier than where it is consumed, in order to hide communication latency. This is somewhat accomplished in our techniques by a combination of choosing the *earliest* min-cut placement (i.e. closest to the source), and the stall-on-use implementation of the consume instruction.

6. Conclusion

This paper presented optimizations that a compiler can perform to reduce the communication instructions for global multi-threaded (GMT) instruction scheduling. These optimizations improve the Multi-Threaded Code Generation (MTCG) algorithm proposed by Ottoni et al. [16], which can support any GMT scheduling algorithm. This paper described COCO, a general framework to optimize register, control, and memory inter-thread dependences. COCO unifies all these optimizations using novel thread-aware data-flow analyses and graph min-cut algorithms. Using a full compiler implementation of COCO and a dual-core simulator built on top of validated Itanium 2 core models, our experiments demonstrated significant results for two previously proposed GMT scheduling techniques, DSWP and GREMIO. COCO reduced the number of dynamic communication instructions by 29.1% on average (up to 73.7%), resulting on an additional 9.2% average speedup (up to 47.6%) for DSWP and GREMIO over the single-threaded codes. Even though our evaluation was restricted to two threads in this paper, we expect the benefits from COCO to be more pronounced when more threads are generated. The reason for this is that, as more threads are created, the larger the number of inter-thread dependences to be respected, and therefore the larger the fraction of communication instructions. Although not explored in this paper, COCO can also be used to improve speculative extensions of GMT scheduling techniques, as well as local MT scheduling techniques. Finally, the techniques developed in this work, including thread-aware data-flow analyses and the use of graph min-cuts to model communication, may be useful in other contexts as well.

Acknowledgments

We thank the entire Liberty Research Group 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. Guilherme Ottoni is supported by an Intel Foundation Ph.D. Fellowship. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

References

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 126–138, 1993.

[2] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: a preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, 1992.

[3] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 1996.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.

[5] 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.

[6] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] 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.

[8] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Trans. Program. Lang. Syst.*, 21(6):1251–1297, 1999.

[9] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[10] 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.

[11] 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 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

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

[13] 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.

[14] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.

[15] G. Ottoni and D. I. August. Global multi-threaded instruction scheduling. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–68, December 2007.

[16] 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*, pages 105–116, November 2005.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[26] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. A. Connors. Chip multi-processor scalability for single-threaded applications. In *Proceedings of the Workshop on Design, Architecture, and Simulation of Chip Multi-Processors*, November 2005.

[27] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007.

[28] Y. Wu and J. R. Larus. Static branch prediction and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 1–11, December 1994.

[29] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 160–169, 2001.

[30] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, 2002.

[31] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.