Partitioning a Lenient Parallel Language into Sequential Threads

Sangho Ha and Sangyong Han

Dept. of Computer Science & Statistics
Seoul National University
Seoul 151-742, Korea

Heunghwan Kim

Dept. of Computer Science
Seowon University
Cheongju 360-742, Korea

Abstract

Multithreading is attractive in a large-scale parallel system since it allows split-phase memory operations and fast context switching between computations without blocking the processor. Performance of multithreaded architectures depends significantly on the quality of multithreaded codes. In this paper, we describe the enhanced thread formation scheme to produce efficient sequential threads from programs written in a lenient parallel language Id⁻ [1]. This scheme features graph partitioning based on only long latency instructions, combination of multiple switches and merges introducing a generalized switch and merge, thread merging, and redundant arc elimination using thread precedence relation. The simulation results show that our scheme reduces control and branch instructions effectively.

1 Introduction

Parallel architectures with a number of off-the-shelf microprocessors should address performance degradation due to increased latency and synchronization overheads[2]. Currently, most architecture designers are trying to conquer these problems by aids of cacheing and multithreading. Multithreading is attractive in a large-scale parallel system since it allows split-phase memory operations and fast context switching between computations without blocking the processor. A thread is usually defined by a sequence of instructions. Its execution may be paused or interrupted as [3,4], may be nonpreemptive[1,5,6,7,8], or may be executed by interleaving among threads[9,10], according to execution model.

A lenient parallel language Id[11] supports nonstrict control constructs or data structures for synchronization, thus allowing massively fine grain parallelism. There are several approaches to generate multithreaded codes from Id programs[1,4,5,6,7,12,13]. It is an important issue how to partition the dataflow program graph, which is an intermediate form of Id programs, to satisfy the following criteria: maximization of parallelism and run-length, and minimization of synchronization costs.

Nikhil[14] doesn't aggregate nodes in dataflow graphs. Each node in the graphs means a thread. He tries to aggregate nodes by optimization, but his aggregation rule is not clear. Iannucci[4] gets partitions by aggregating nodes with the same set of input dependences, dealing with long latency instructions as split-phase transactions. But he is not concerned with thread merging. Schauser[7] partitions the graphs using dependence set and dominance set. Also, he tries to lengthen the size of thread through thread merging. In his algorithm, however, switches and merges give a separation constraint on partitioning, thus resulting in a large number of small threads. In his advanced work[13], thread length can be enhanced by global analysis. However, the partitioning constraint given to switches and merges still remains, and it is not reported that the global analysis scheme was implemented. Hoch's algorithm[6] is similar to that of Schauser. He tries to combine *then-* and *else- arm* of conditionals using control vector. But it may cause additional branch instructions, which increase with the nested level of conditionals.

In this paper, we describe the thread formation scheme implemented on the multithreaded architecture DAVRID[15]. We first define the DAVRID graph as an intermediate form, and partition the graph using dependence set[4] with giving no constraint to switch and merge. Then, we merge the related threads using thread precedence relation, causing no additional branch instructions. Thus, in terms of conditional execution, our scheme may be considered as an intermediate approach between Schauser's and Hoch's. In addition, we combine multiple switches and merges by the generalized switch and merge newly introduced, and eliminate redundant arcs using thread precedence relation.

In section 2, we describe an execution model of programs on DAVRID. In section 3, we describe the DAVRID graph where our works begin with. In section 4, we describe the process of forming multithreaded codes from the DAVRID graph in detail. In section 5, we analyze thread codes through simulation results over several benchmarks and compare our scheme with Schauser's. Finally, in section 6 we remark conclusions and future works.

2 Execution Model

Our program execution model is based on the hybrid of dataflow model and von Neumann model, called multithreaded execution model. Over thread level, we exploit natural synchronization and massive parallelism by adopting dataflow computing rule. Within a thread, we exploit locality of computations by the use of registers and caches by adopting von Neumann computing rule. A program consists of several code blocks, where a code block denotes a context and usually corresponds to a function or a loop. A code block consists of multiple interrelated threads. Each thread is defined by a sequence of instructions totally ordered. Also, it doesn't wait for any response for remote requests. So, senders and receivers of remote requests are placed in separate threads.

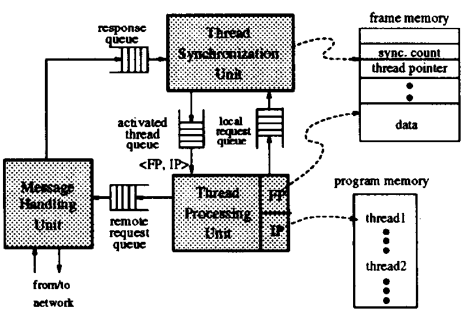

Figure 1: DAVRID Abstract Machine

The abstract machine of DAVRID is shown in Figure 1. We describe how threads are executed on DAVRID below. The machine consists of three processing units: thread processing unit, thread synchronization unit, and message handling unit. Each unit has its own local memory and may work independently. Note that Figure 1 is an abstract view of the real DAVRID machine[15].

The thread processing unit gets a continuation denoting information about a thread from the activated thread queue, and executes the thread sequentially. Remote requests are sent to the message handling unit through the remote request queue or the local request queue, depending on the destination address of the requests. Messages contain a continuation as well as values. A continuation is represented by ⟨ *frame pointer,instruction pointer* ⟩ : a *frame pointer* is the base address of the frame containing data to be referred in frame memory, and an *instruction pointer* is the starting address of the thread in program memory. Note that frames are allocated for function calls. Thread execution is nonpreemptive. As soon as a thread terminates, the thread processing unit gets a continuation from the activated thread queue. The thread synchronization unit performs synchronizations among threads. It gets a message from the response queue, decrements the synchronization count of the thread indicated by the message, and stores a value in the frame slot specified by the message if exists, and activates the thread if the synchronization count comes to zero. The continuation of the activated thread is put on the activated thread queue. Note that the synchronization count of a thread is initialized by the number of data or signals needed for the activation of the thread. The message handling unit functions as a message router. Also, it manages the global structured memory and load balancing.

3 DAVRID Graph

The entire compilation process for DAVRID is described in [1]. The compiler uses the program graph[16] as an intermediate form between the front end and back end. Program graphs are transformed into machine graphs for DAVRID. The machine graph, called as the DAVRID graph afterwards, has well-defined operational semantics and can be executed directly. Our partitioning scheme begins with this DAVRID graph. Note that the DAVRID graph has the same level of abstraction as dual graph[7], P-RISC graph[5], or HDG[6].

The meaning of the DAVRID graphs is only given by the dataflow firing rule. In multithreaded execution model, control flow is explicit and data flow is implied by registers or frame slots. To overcome this gap, Schauser[7] introduces dual graph in which control and data flows are both explicit. In contrast, we view that the directed arcs between nodes in the DAVRID graphs implie control flow as well as data flow, *i.e.* an arrival of values on arcs means that a

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

control also arrived. In our execution model, since an event may cause a synchronization count to be decreased and a value to be stored in the specified frame slot, data and control flow are combined as well in terms of operational semantics. Note that arcs in the DAVRID graphs have the same semantics as those of HDG[6] where arcs denote data or control dependencies between nodes.

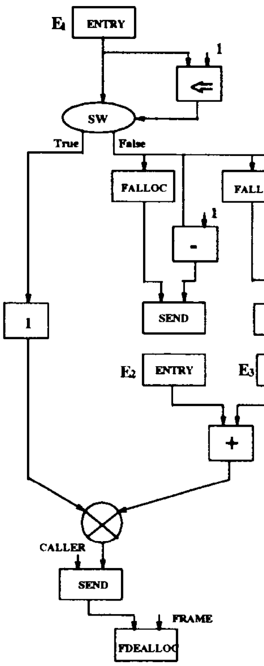

Figure 2: A DAVRID Graph for Fibonacci Program

Figure 2 shows the DAVRID graph for the following fibonacci program written in Id$^-$[1], which is a functional core of Id.

fib = {fun n = (if ($n \leq 1$) then 1
else fib (n-1) + fib (n-2))};

Entry nodes are usually used for interface between callers and callees. In Figure 2, E_1 receives an argument from a caller. E_2 and E_3 receive a value from a callee, respectively. Entry nodes are semantically similar to inlets in [5,7], but different in terms of operations: That is, while inlets receive a message and process it directly, entry nodes get only the effect after messages are processed. Note that messages are processed by the thread synchronization unit as mentioned in section 2.

FALLOC and FDEALLOC are the DAVRID machine instructions denoting requests for allocation and deallocation of frames, respectively. If values arriving at E_1 are greater than 1, the graph allocates two frames by FALLOCs. An arrival of values on the output arc of the FALLOC node means the following: 1) the address of the requested frame is available. 2) a synchronization event required to send an argument to a callee occurred. So, the output arc conveys a control as well as a value.

4 Proposed Partitioning Scheme

Several issues must be taken into consideration on graph partitioning: 1) maximization of exploitable parallelism, 2) maximization of run length, 3) minimization of explicit synchronization, 4) deadlock avoidance, and 5) maximization of machine utilization[4]. In our partitioning scheme, the DAVRID graphs are first partitioned based on only long latency instructions, thus preserving all exploitable parallelism. Long latency instructions such as fetch-like instructions in [4] may cause unpredictable delay due to network routing or deferred reads on I-structure memory. This partitioning method is proved to be safe in [4]. Partitions are merged into larger ones on the basis of thread precedence, to satisfy the second and third conditions. Also, we decrease the number of explicit synchronization events by removing redundant arcs based on thread precedence relation, to satisfy the third condition. In addition, we combine multiple switches and merges by introducing a new generalized switch and merge, and perform thread refinement to support non-strictness of Id$^-$.

4.1 Basic Partitioning

The DAVRID graphs consist of nodes and arcs where a node represents an instruction and an arc represents data and control flow. Arcs in the graphs may be local or remote. Remote arcs are originated from long latency instructions while local arcs are not. Long latency instructions are the instructions whose execution time is long and unpredicatable.

We partition the DAVRID graphs by cutting remote arcs logically, thus making it possible to deal with long latency instructions as split transactions. This is performed by using dependence set developed by Iannucci[4]. In [7], graphs are partitioned using dependence set and dominance set, where dependence set is a variant of Iannucci's dependence set since labels and merges are dealt with as entry instructions

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

starting a new partition. Note that a label is placed on each output of a switch, indicating a separation constraint. So, the predicate and each arm of a conditional are placed in separate threads, respectively.

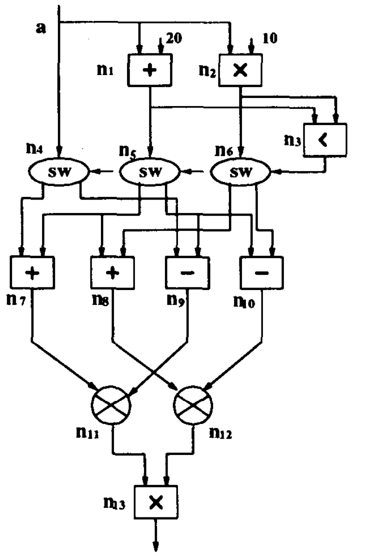

Figure 3: Example of Basic Partitioning

In contrast, our partitioning scheme doesn't have any restriction except for long latency instructions. So, any conditional with no long latency instruction can be transformed into a single thread. Thus, our scheme provides the ability to form threads across basic blocks of conventional branches. For example, we consider the DAVRID graph in Figure 3, having no remote instruction. Since all entry points of the graph, n_1, n_2, and n_4 use the value of a, all nodes in the graph have the same input dependence set. While the graph is not partitioned under our scheme, it is partitioned into eight groups under Schauser's: $\{n_1,n_2,n_3,n_4,n_5,n_6\}$, $\{n_7\}$, $\{n_8\}$, $\{n_9\}$, $\{n_{10}\}$, $\{n_{11}\}$, $\{n_{12}\}$, $\{n_{13}\}$, where each set denotes a partition.

4.2 Combination of Switch and Merge

There may exist multiple switches or multiple merges determined by the same predicate in a thread. For example, three switches and two merges in Figure 3 are placed in the same partition for our scheme. These are redundant because they are determined by the same predicate. Redundant switches and merges can be eliminated with no loss of information.

Schauser[7] suggests the combination method of switches and merges by the graph transformation, causing additional signal trees. Also, his scheme has not been reported to be implemented. We will combine multiple switches and merges without any difficulty by introducing a generalized switch and merge.

A generalized switch is defined by a switch that can have multiple inputs(except for a control input) and multiple outputs for each arm, being different from a conventional switch having one input and two outputs. As shown in (a) of Figure 4, the i-th input of the generalized switch corresponds to the i-th output for *then-arm* and *else-arm*, respectively. The semantics of the generalized switch is that the i-th input value is conveyed to the i-th output of *then-arm* or *else-arm* depending on the value of the control input. So, the i-th input and the i-th output for each arm are interpreted as a conventional switch. Thus, a generalized switch with n inputs and $2n$ outputs has an effect of combining n conventional switches.

Similarly, a generalized merge is defined by a merge that can have multiple inputs and multiple outputs. The i-th output of generalized merge corresponds to the i-th input for *then-arm* and *else-arm*, respectively as shown in (b). The semantics of the generalized merge is that the i-th output value is conveyed from the i-th output of *then-arm* or *else-arm*. So, the i-th two inputs and the i-th output are interpreted as a conventional merge. Thus, a generalized merge with n outputs and $2n$ inputs has an effect of combining n conventional merges.

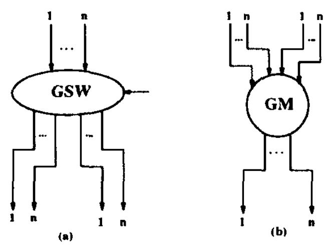

Figure 4: Generalized Switch and Generalized Merge

Figure 5 shows how three switches and two merges in the graph of Figure 3 are combined using generalized switches and merges. We see that one input and two outputs for i-th ($1 \leq i \leq 3$) switch in Figure 3 correspond to the i-th input, the i-th output(for *then-arm*), and the $(i+3)$-th output(for *else-arm*) of GSW, respectively. Also, we see that two inputs and one output for i-th($1 \leq i \leq 2$) merge correspond to the i-th input(for *then-arm*), the $(i+2)$-th input(for *else-arm*), and the i-th output of GM, respectively. We can derive easily that the two graphs have the same meaning.

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

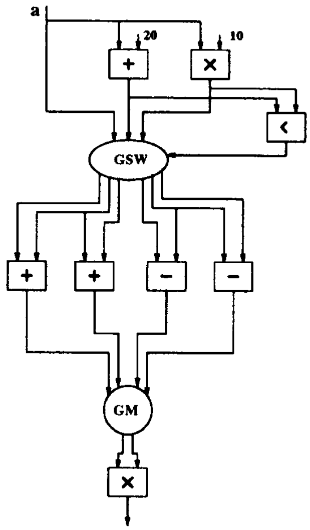

Figure 5: Combination of Switches and Merges

4.3 Thread Refinement

Non-strictness of Id^- may make it impossible to determine the execution order of a function statically since results of the functions may be fed back in as its arguments. This order may depend on the dynamic context in which the function appears. We consider an example for non-strict conditionals presented in [12] as following.

```
conditional_example = { fun x =
      { a = if x > 0 then bb else 3;
        b = if x < 0 then aa else 4;
        aa = a + 5;
        bb = b + 6;
        c = a + b;
       in c}};
```

We can see that the cycles arise in the corresponding graph, *i.e.* for negative value of x, a and aa must be evaluated before b and bb, while for positive values of x the reverse is true. Since execution order of aa and bb depends on the value of x, it can not be determined statically. Thus, codes that compute aa and bb must be placed in separate threads. However, our partitioning scheme will form the graph for the above conditional_example into a single thread, since there is no long latency instruction in the graph. Obviously, it is not correct. We can solve this problem by refining the graph, *i.e.* by cutting each arm of switches in the graph. Our compiler can recognize easily the graphs with non-strict conditionals by examining cyclic edges.

We observed that most application programs don't include non-strict conditionals. So, we believe that our approach to thread formation is practical.

4.4 Thread Merging

The second goal of our partitioning scheme is to maximize run length. We will get this goal by merging statically a collection of related threads into one. Our thread merging is based on thread precedence relation. We first construct thread precedence graph(TPG) where each node and arc denote a thread and precedence relation between threads, respectively. Then, we merge threads by examining precedence between threads from TPG. We define thread precedence relation using input dependence set(IDS) below. Note that all instructions within a thread have the same IDS.

Definition 1 *Given two threads T_1 and T_2, T_1 precedes T_2 only if IDS(T_1) $\subset$ IDS(T_2), meaning that T_2 can start to execute after T_1 terminates.*

Definition 2 *Given two threads T_1 and T_2, T_1 directly precedes T_2 only if T_1 precedes T_2 and there are arcs from T_1 to T_2.*

Definition 3 *Given two threads T_1 and T_2, T_1 indirectly precedes T_2 only if T_1 precedes T_2 and there is no arc from T_1 to T_2.*

Definition 4 *Given two threads T_1 and T_2, an arc between T_1 and T_2 is remote only if at least one long latency instruction of T_1 is connected with an instruction of T_2; otherwise, the arc is local.*

Definition 5 *Given a thread T, threads directly preceding T are fathers of T, and threads preceded directly by T are children of T.*

Definition 6 *Given a thread T, threads preceding T are ancestors of T, and threads preceded by T are descendants of T.*

Definition 7 *A thread T is simple only if it consists of only instructions that don't cause any effect external to T. That is, simple threads don't store any value on I-structure as well as don't make any request whose execution may cause long latency or unpredicatable delay. Note that the entry instruction is dealt with as a simple thread.*

In addition, we give another constraint on thread merging: candidate threads to be merged must be simple. If non simple thread T is merged, execution of some descendants of T may be delayed since they may wait for another events that are unnecessary for them to be activated. Also, thread merging may not be safe due to non-strictness.

Thread Merging Rule: *Given two collection X and A of threads satisfying the following four conditions: i) all threads in A are simple. ii) each thread in X is directly preceded by all threads in A. That is, all elements in A are fathers of all elements in X. iii) all children of each thread in A are included in X. iv) there exists inclusion relation between IDS(T_i) and IDS(T_j) for each T_i, $T_j \in X$.*

Then we can select a thread T with the least IDS in X such that $IDS(T) = \cap_{x \in X} IDS(x)$, and we can merge T and all threads in A.

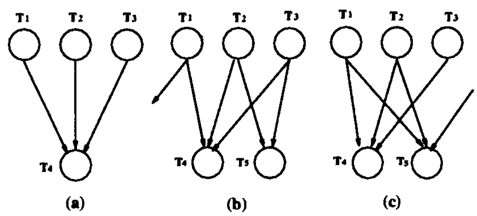

Figure 6: Examples of Our Merging Scheme

Figure 6 shows how threads are merged by our merging rule. We assume T_1, T_2, and T_3 are simple. In a), A = $\{T_1,T_2,T_3\}$ and X = $\{T_4\}$. Since the graph in a) satisfies all preconditions, we can merge T_1, T_2, T_3, and T_4. It causes a loss of parallelism, but is effective in that T_1, T_2, and T_3 are all simple. We use 'effectiveness', considering trade-offs between parallelism and synchronization overheads. Each T_i ($1 \leq$ i ≤ 3) has a little complexity in computation and doesn't affect external codes outside its own thread since it is simple. We also note that in multithreaded execution, synchronization overheads among threads are much larger than computation complexity of simple threads. So, merging of T_1, T_2, T_3 is rather advantageous than preserving parallelism, which removes synchronization overheads among three threads. In b), A = $\{T_2,T_3\}$ and X = $\{T_4,T_5\}$. Since IDS(T_5) $\subset$ IDS(T_4), we can merge T_2, T_3, and T_5. Note that in this case, any thread may not be merged under Schauser's merging rules[7,13]. In c), there is no inclusion between IDS(T_4) and IDS(T_5). So, we can't merge T_1, T_2 with T_4 or T_5 with no loss of parallelism. Note that T_5 must wait for T_3 to be activated unnecessarily if T_1, T_2, and T_4 are merged.

We discuss static and dynamic cycles[4] on the graph. *A static cycle* is defined by a directed cycle of threads for which no schedule of the threads can terminate. For example, when two threads, T_i, T_j form a static cycle, T_i and T_j require the output of an intermediate instruction of the other thread, which will be executed before the corresponding thread terminates, respectively. When execution of threads are nonpreemptive, T_i and T_j can never be activated, resulting in deadlock. A dynamic cycle is caused by operations on I-structure, which introduce the notion of dynamic arcs between producers and consumers. *A dynamic cycle* is defined by a directed cycle of threads, augmented with the input-specific dynamic arc, for which no schedule of threads can lead to terminate. Dynamic cycles can be avoided by placing the consumers and the producers whose execution orders are input-dependent in separate threads. Note that graph partitioning based on dependence set doesn't cause any static and dynamic cycle[4].

Theorem 1 *Our thread merging rule is safe.*

Proof: We will prove theorem 1 by showing that our thread merging rule doesn't create any static or dynamic cycle on the graph. Since our graph partitioning is based on dependence set, there doesn't exist any static and dynamic cycle on TPG from [4]. So, we will only show that no static and dynamic cycle can occur by applying our merging rule.

Let A and X be two sets satisfying all preconditions of the merging rule, and let T be an element of X selected by the rule, *i.e.* $IDS(T) = \cap_{x \in X} IDS(x)$. For each element T_α in A, T_α is simple by the first condition. So, all arcs from T_α to T are not remote. Thus, there is no relation of producer and consumer by I-structure between T_α and T. So, any dynamic arc from T_α to T is not created. Therefore, any dynamic cycle can not occur by merging T_α and T.

Now, we will show that no static cycle can occur. Let G be a collection of all elements in A and T. We define IDS(G) to be $\cup_{g \in G}$IDS(g). Then, IDS(G) = IDS(T) since all elements in A precede T. For any ancestor T_α of each element T_i in A, IDS(T_α) $\subset$ IDS(T_i). Since IDS(T_i) $\subset$ IDS(G), IDS(T_α) $\subset$ IDS(G) by transitivity. Therefore, all ancestors of A precede G (i). Also, for any descendant of T_β of T, IDS(T) $\subset$ IDS(T_β). Since IDS(G) = IDS(T), IDS(G) $\subset$ IDS(T_β). Therefore, G precedes all descendants of T (ii). Finally, when $|X| > 1$, for any element T_γ in X($T_\gamma \neq$ T),

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

since $IDS(T) = \cap_{x \in X} IDS(x)$, IDS(T) $\subset$ IDS(T_γ). So, IDS(G) $\subset$ IDS(T_γ) since IDS(T) = IDS(G). Therefore, G precedes T_γ (iii). By i), ii), and iii) the merging rule doesn't create any cycle on TPG (End).

Hoch[6] tries to combine *then-* and *else- arm* of conditionals using control vector. We consider the graph in Figure 7. The graph is basically partitioned into at least four groups: $T_1 = \{n_1\}$, $T_2 = \{n_2\}$, $T_3 = \{n_3\}$, $T_4 = \{n_4\}$, $T_5 = \{n_5, n_6\}$. These all groups are merged into one in his merging scheme. But the result needs two additional branch instructions, which is due to merging T_1, T_2, and T_3. Note that T_1, T_2, and T_3 may be also merged under our merging algorithm. However, they can be refined to avoid additional branch instructions, considering trade-offs between parallelism and synchronization overheads. In that case, T_1, T_2, and T_3 become a separate thread, respectively. Note also that these additional branch instructions increase with the level of nested conditionals.

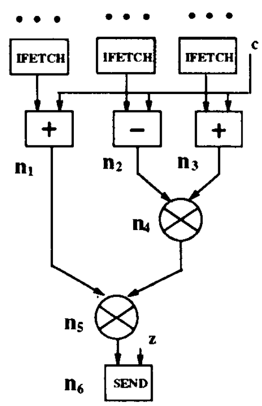

Figure 7: Example of Hoch's Merging Scheme

4.5 Synchronization Count of Threads

The number of events required for activation of threads is called synchronization count. The synchronization counts of threads, which is computed by the number of input arcs of threads on the graphs, may be overestimated by redundant events. So, synchronization overheads must be reduced by eliminating redundant arcs in the graphs. Redundant arc elimination in [7] is quite primitive. Moreover, it may be expensive and is limited since it is required to search graphs to identify candidates for elimination. We will eliminate redundant arcs based on thread precedence relation with no difficulty. Note that any remote arc is not considered as a redundant arc since it denotes an unpredicatable synchronization event.

Our algorithm of computing the synchronization count of a thread T is as follows.

```
Let S be a set consisting of synchronization events
Initialize S by an empty set
FOR (each input arc a of T)
   Let the father of T wired by a be f
   IF (a is remote)
      insert f into S
   ELSE IF (there exist s(∈ S) s.t. s precedes f)
      replace s by f
   ELSE
      insert f into S
END FOR
Set the synchronization count of T by the size of S
```

After initializing a set S, the algorithm examines each input arc a of T on TPG. If the a is remote, insert f into S; otherwise, examine if f precedes any element in S. If f is preceded by some element s in S, replace s by f; otherwise, insert f into S only if it doesn't precede any element in S. That is, for each father t_α of T, there is no t_β in S such that the arc between t_α and t_β is local, and IDS(t_β) $\subset$ IDS(t_α). The size of S is computed as the synchronization count of T.

Note that all fathers of T which are locally wired to T and precede some elements in S are not considered on computing the synchronization count of T. That is, control flows from a father f of T which is not included in S are subsumed by control flows of an element in S which is preceded by f. So, Control flows from f to T are redundant. Thus, they can be removed correctly. Also, there may be multiple control flows between $s(\in S)$ and T on the DAVRID graph. If there is at least one remote arc among them, all local arcs among them are removed; otherwise, only one local arc is selected. In this way, all redundant control flows to T can be removed. Therefore, the synchronization count of T computed by our algorithm is minimal.

In Figure 8, where a dotted arrow denotes an indirect precedence and all input arcs to T are local, we explain how redundant arcs to T_4 are eliminated. Fathers of T_4 are T_1, T_2, T_3. We see that T_1 indirectly precedes T_2, which directly precedes T_3. So, control flows from T_1 and T_2 to T_4 are subsumed by control flows from T_3. Thus, we don't consider T_1 and T_2 on computing the synchronization count of T_4. Also, we select only one control flow from T_3. Therefore the synchronization count of T_4 is computed by one.

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

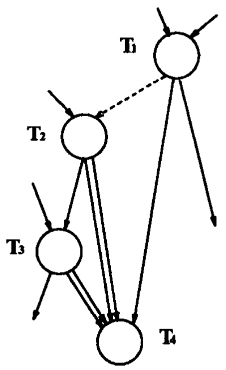

Figure 8: Example of Redundant Arc Elimination

However, it has the value of five if the redundant arc elimination is not performed.

5 Analysis

This section describes an analysis on the quality of the DAVRID codes produced by our compiler through simulation results over several benchmarks. Note that the thread codes used in this analysis are written in the DAVRID parallel machine language[15]. Three benchmark programs are used: integer sum, lookup, wavefront. Integer sum computes the sum of all integers between 1 and 65,535 using the method of parallel summation. Lookup performs binary searching for 10,000 elements. Wavefront computes a sequence of matrices where each element of the new matrix is computed by the sum of its neighbors to the left and the upper in the old matrix. The size of the matrix is 100×100.

We focus on the comparison between two thread formation schemes: our scheme and Schauser's. Between two schemes, there is a critical difference in dealing with conditionals: whether to give separation constraint on switch and merge or not. So, only benchmarks including conditionals in programs are selected. Also, all loops in benchmarks are sequentially executed without unfolding, which makes comparison between two scheme more exact.

Table 1 compares the characteristics of threads between two schemes for each benchmark. 'No. of threads' denotes the total number of threads generated for each program, and 'Thread length' denotes the average size of threads, computed by $\frac{N}{T}$ where N is the total number of instructions, and T the total number of threads. 'Synch count' denotes the average synchronization count of threads, computed by $\frac{S}{T}$ where S is the summation of synchronization counts over all threads. Our scheme has fewer threads and synchronization counts, while it has longer thread length. This is because Schauser's scheme creates a large number of small threads due to the separation constraint on switches and merges. Especially, for lookup in which there are two conditionals in a *while* expression, his scheme has about 3.5 times number of threads as our scheme. Also, thread merging is rather restricted in his scheme because of his less powerful merging rule and no combination of switches and merges. On the other hand, the synchronization count of integer sum is a little larger than in his scheme. This is because there are a number of threads with one value of synchronization count, which are caused due to the separation constraint. For other benchmarks, our scheme has smaller synchronization counts. Note that average thread length in our scheme is nearly twice that of typical branch distances.

Table 2 shows relative instruction frequencies dynamically measured between two schemes for each benchmark. Instructions are classified into seven categories: synchronization, memory management, structured memory access, ALU, load/store, branch, and others. Synchronization consists of all instructions performing synchronization events. Memory management consists of operations performing allocation/deallocation of frames or heaps. Load/store means load/store operations on boundaries of threads. Others include house-keeping operations such as NEXT executed at the end of thread and operations to initialize or reset synchronization counts of threads. Note that NEXT makes the thread processing unit get another continuation from the activated thread queue. Usually, synchronization, load/store, and others form the control portion of total codes, which is inevitable overheads due to multithreading. On the other hand, memory management, structured memory access, ALU, and branch form the computation portion.

In Table 2, the frequency of control and branch is larger in Schauser's scheme, while the frequency of computation is equal for two schemes. We see the following facts: for the execution of each thread, it is required to load/store values from and to frame slots for the boundary values of the thread and perform a NEXT. Also, it is required to reset the synchronization counts of all threads composing loop bodies on iteration boundaries. Thus, as wee can see in Table 1, in his scheme the larger number of threads with the smaller size and larger synchronization count cause the larger

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

Table 1: Thread Characteristics

		integer sum	lookup	wavefront
No. of threads	Schausers'	25	21	18
	Ours	16	6	11
Thread length	Schausers'	5.4	6.0	8.7
	Ours	6.1	11.8	11.5
Synch count	Schausers'	1.6	1.6	1.9
	Ours	1.8	1.2	1.2

Table 2: Instruction Frequencies

		Synch	Memory	SM Access	ALU	Load/Store	Branch	Others
Integer sum	Schausers'(1)	329649	254	0	131577	594906	197244	925074
	Ours (2)	1397	254	0	131577	331493	131454	70375
	(1)/(2)	236.0	1.0		1.0	1.8	1.5	13.2
Lookup	Schausers'(1)	278	1	52	184	573	151	925
	Ours (2)	39	1	52	184	336	65	346
	(1)/(2)	7.1	1.0	1.0	1.0	1.7	2.3	2.7
Wavefront	Schausers'(1)	110301	1.0	59204	306120	517541	40401	579354
	Ours (2)	40301	1.0	59204	306120	368230	40401	339337
	(1)/(2)	2.7	1.0	1.0	1.0	1.4	1.0	1.7

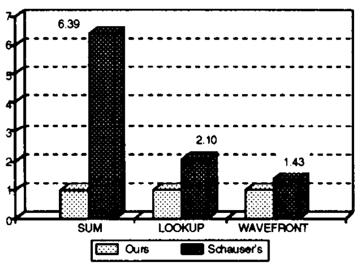

Figure 9: Relative Execution Time

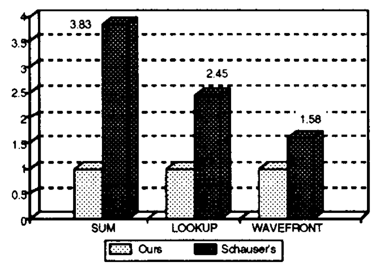

Figure 10: Relative Instruction Frequency

frequency of control operations correspondingly. We also can see that our scheme has fewer branches for integer sum and lookup. This is due to the fact that switch combination reduces the number of branch instructions effectively.

We can see a strong relationship between execution time and the frequency of instructions in Figure 9-10: execution time increases in proportion to the frequency of control and branch instructions. Also, we see that for each benchmark, the ratio of execution time corresponds similarly with the ratio of the frequency of control and branch. For integer sum, however, there is some difference between two ratios. That is, the ratio of the execution time between two schemes is 6.39, but the ratio of the frequency of control and branch is 3.83. We see that the ratio of the synchronization frequency of integer sum in Table 2 is very large, compared with other categories. Also, synchronization instructions performed by a message handler require more execution time than other instructions. So, the frequency of synchronization instructions affects execution time of programs more deeply.

6 Conclusion

In this paper, we described an enhanced thread formation scheme to produce efficient sequential threads from Id^- programs. Simulation results over several benchmarks show that the portion of control and branch operations of total codes are deeply affected by thread formation schemes. The portion of control is essentially overheads due to multithreading. Unless the overheads are controlled effectively, good performance can't be expected in multithreaded architectures. Our scheme reduces the portion of control and branch effectively, compared with Schauser's.

Run length can be enhanced by thread scheduling. TAM[8] performs compiler-controlled thread scheduling, exploiting locality of computations. As indicated

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE

in [5], however, it may have considerable scheduling overheads since it manipulates a complicated data structure and performs atomic operations frequently. Also, speculative register management, which allows quanta to grow up dynamically, is not clear. Note that a quantum in TAM is defined as an unit of scheduling which consists of the consecutive execution of several threads. P-RISC[5] tries to reduce these scheduling overheads by message based scheduling, where a set of threads activated during a message handling forms a quantum as TAM. But it can't exploit locality sufficiently. Also, it may delay message processing by the use of a general interrupt handler. We are considering a thread scheduling scheme to implant the concept of quantum effectively on the DAVRID machine. In DAVRID, since the synchronization unit exclusively processes all messages carrying synchronization events as inlets in TAM, the thread processing unit don't experience any scheduling overhead as TAM and P-RISC. Also, it can exploit locality of computations by the local request queue.

The size of thread can be substantially lengthened by incorporating data parallelism. If data in a global structured memory are placed in local memory in advance before execution, all operations to refer these values need not to be dealt with as long latency instructions any more. So, in our partitioning scheme that considers only long latency instructions as a separation constraint of the graph, thread length can be substantially enhanced. Also, thread length can be enhanced by global analysis[13], which makes it possible to merge multiple entry nodes by propagating dependence information across control boundaries and function calls. However, as pointed out in [13], it is necessary to consider trade-offs among thread size, parallelism, communication, and synchronization overhead for optimal thread execution.

References

[1] E.H. Rho, S.H. Ha, S.Y. Han, and et al, "Compilation of a Functional Language for the Multithreaded Architecture: DAVRID", Int'l Conf. on Parallel Processing, Aug. 1994.

[2] Arvind, R.A. Iannucci, "Two Fundamental Issues in Multiprocessing", MIT CSG Memo 226-5, 1986.

[3] L. Bic, "A Process-Oriented Model for Efficient Execution of Dataflow Programs", J. of Parallel and Distributed Computing, Vol. 8, pp. 42-51, 1990.

[4] R.A. Iannucci, "Parallel Machines: Parallel Machine Languages", Kluwer Academic Publishers, 1990.

[5] R. S. Nikhil, "A Multithreaded Implementation of Id using P-RISC Graphs", Int'l Proc. of 6th Annual Workshop on Languages and Compilers for Parallel Computing", 1993.

[6] J.E. Hoch, et al, "Compile-time Partitioning of a Non-strict Language into Sequential Threads", Int'l Proc. of the 3rd IEEE Sympo. on Parallel and Distributed Processing, pp.180-189, 1991.

[7] K. E. Schauser, D. E. Culler, T. Eicken, "Compiler-Controlled Multithreading for Lenient Parallel Languages", Conf. on Functional Programming Languages and Computer Architecture, 1991.

[8] D.E. Culler, S. C. Goldstein, et al, "TAM-A Compiler Controlled Threaded Abstract Machine", J. of Parallel and Distributed Computing, Vol. 18, pp. 347-370, July 1993.

[9] G.M. Papadopoulos, D.E. Culler, "Monsoon: an Explicit Token-Store Architecture", 17th Annual Int'l Sympo. on Computer Architecture, 1990.

[10] B. Smith, "The Architecture of the HEP, In Parallel MIMD Computation: HEP Supercomputer and its Applications, MIT Press, 1985.

[11] R.S. Nikhil, "Id(Version 90.1) Reference Manual, Technical Report MIT CSG Memo 284-2, 1991.

[12] K.R. Traub, "Sequential Implementation of Lenient Programming Languages", Technical Report, MIT/LCS/TR-417, Sept. 1988.

[13] K.R. Traub, D.E. Culler, and K.E. Schauser, "Global Analysis for Partitioning Non-Strict Programs into Sequential Threads", Conf. on LISP and Functional Programming", 1992.

[14] R.S. Nikhil, "The Parallel Programming Language Id and its Compilation for Parallel Machines", MIT CSG Memo 313, 1990.

[15] S.H. Ha, J.H. Kim, and et al, "A Massively Parallel Multithreaded Architecture: DAVRID", IEEE Int'l Conf. on Computer Design, Oct. 1994.

[16] K.R. Traub, "A Compiler for the MIT Tagged Token Dataflow Architecture", Technical Report TR-370, MIT CSG, 1986.

Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)
1060-3425/95 $10.00 © 1995 IEEE