A General Compiler Framework for Speculative Multithreaded Processors

Anasua Bhowmik and Manoj Franklin

Abstract—Speculative multithreading (SpMT) promises to be an effective mechanism for parallelizing nonnumeric programs, which tend to have irregular and pointer-intensive data structures and complex flows of control. Proper thread formation is crucial for obtaining good speedup in an SpMT system. This paper presents a compiler framework for partitioning a sequential program into multiple threads for parallel execution in an SpMT system. This framework is very general and supports speculative threads, nonspeculative threads, loop-centric threads, and out-of-order thread spawning. It is therefore useful for compiling for a wide variety of SpMT architectures. For effective partitioning of programs, the compiler uses profiling, interprocedural pointer analysis, data dependence information, and control dependence information. The compiler is implemented on the SUIF-MachSUIF platform. A simulation-based evaluation of the generated threads shows that the use of nonspeculative threads and nonloop speculative threads provides a significant increase in speedup for nonnumeric programs.

Index Terms—Multithread processor, scheduling and task partitioning, compilers.

1 Introduction

One of the definitive challenges in computer science and engineering over the last several decades has been reducing the completion time of a single computational task. The primary means of reducing execution time, besides decreasing the clock period and the memory latency, has hinged on exploiting the inherent parallelism present in programs. Parallelization of programs has been a success for scientific applications, but not for nonnumeric applications. The emergence of the *speculative multithreading* (SpMT) model [2], [10], [12], [13], [16], [18], in the last decade has provided the much awaited breakthrough for nonnumeric applications. The hardware support that this model provides for speculative thread execution makes it possible for the compiler to parallelize *sequential* applications without being constrained by the data dependences and control dependences present in the program.

This paper presents and evaluates a general compiler framework that we have developed for partitioning sequential programs (especially nonnumeric programs) into multiple threads for parallel execution in SpMT systems. Traditional compiler work in parallelization has targeted scientific applications, and has focused mainly on loops that have predefined loop bounds and operate on regular data structures such as arrays. Nonnumeric applications, by contrast, often have loops with large loop bodies, complex flows of control, loop-carried dependences, and loop bounds that cannot be determined statically. Also, unlike scientific applications, nonnumeric applications access irregular data structures with an abundance of pointers and sometimes spend considerable time outside loops. Many of the parallelization techniques used for scientific programs cannot therefore be directly applied to nonnumeric programs.

To parallelize nonnumeric programs effectively, our compiler considers all parts of the program—loop regions as well as nonloop regions—and uses control dependence information and profile information. Its main features are listed below:

- Extracts parallelism from loop regions as well as nonloop regions.
- Very general thread model (allowing it to be used for various SpMT processors).
  - Control-speculative threads as well as control-nonspeculative threads.
  - Multiple spawning points *anywhere* inside the spawning thread.
  - Out-of-order spawning of threads.
  - Shared register name space and shared memory address space for threads.
- Explicitly exploits control independence information while forming threads.
- Performs interprocedural analysis and considers data value predictability during thread formation.

Simulation studies with our SpMT compiler have led to the following conclusions:

- Nonnumeric programs can be successfully partitioned into SpMT threads.
- For nonnumeric programs, it is important to exploit parallelism from both the loops and the nonloop regions.
- It is important to model interthread data dependences carefully to exploit parallelism in SpMT systems.

The rest of this paper is organized as follows: Section 2 provides background information on SpMT and existing

- *A. Bhowmik is with the Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India. E-mail: anasua@csa.iisc.ernet.in.*
- *M. Franklin is with the Electrical and Computer Engineering Department and UMIACS, University of Maryland, College Park, MD 20742. E-mail: manoj@eng.umd.edu.*

Manuscript received 22 Apr. 2003; revised 9 Oct. 2003; accepted 18 Dec. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0063-0403.

work on SpMT compilation. Section 3 describes our SpMT compiler framework. Section 4 discusses utilizing data value predictability information during thread generation. Section 5 presents an experimental evaluation of our compiler. Section 6 presents our conclusions.

2 Speculative Multithreading (SpMT)

The central idea behind SpMT is to execute in parallel threads obtained from a (sequential) program. A number of SpMT models have been proposed, including multiscalar [10], superthreading [18], DMT [2], clustered speculative multithreading [13], and CMP [12], [16]. Speculative multithreading enables parallelization of applications, despite any uncertainty about (control or data) dependences that may exist between the parallel threads. An SpMT processor generally supports both control speculation and data speculation. The hardware speculates on dependences and recovers whenever a speculation is found to be incorrect. This allows the SpMT compiler to do optimistic speculation, thereby improving the performance. Below, we briefly review some of the important aspects of SpMT; a detailed description is available in [6].

2.1 Spawning Strategies

Spawning refers to creating dynamic threads and is analogous to the fork mechanism used in conventional parallel processing; the thread that initiates a spawning is called the *spawning thread*. Spawned threads are assigned to idle processing elements (PEs) for execution.

Spawning Point: An important issue in an SpMT model concerns the points in a thread from where the spawning of other threads can be initiated. Two possibilities exist for the location of the spawning point: 1) at the beginning of the spawning thread and 2) anywhere in the spawning thread. The first case tries to maximize PE utilization by minimizing the time an idle PE waits for a thread to be activated in it. A potential drawback with this approach is that a speculative thread may be spawned prematurely without considering enough runtime information. In the second approach, spawning can be done from anywhere within a thread. This allows the spawning to be delayed, if required, until a particular branch or data dependence gets resolved.

Out-of-Order Spawning of Threads: With out-of-order spawning, threads are not necessarily spawned in program order. That is, a sequentially earlier thread may be spawned after spawning a subsequent thread. In such a situation, some threads may occasionally need to be preempted so as to execute sequentially earlier threads that were spawned later. It is also possible to have SpMT models with limited out-of-order spawning. In the case of an out-of-order depth of 1, for instance, after spawning a thread, at most one predecessor thread can be spawned.

2.2 Nature of Threads

Loop-Centric Threads versus Nonloop Threads: Loop iterations are an obvious candidate for forming threads,and have been the traditional target of parallelization. Each iteration can be specified as a separate thread. The only form of control dependences shared between multiple threads of this kind are loop termination branches, whose outcomes are generally biased toward loop continuation. In nonnumeric programs, many of the loops have loop-carried dependences and, so, it is important to form threads from nonloop regions also.

Control Speculative versus Control Nonspeculative Threads: Speculative spawning—where the existence of the spawned thread is control dependent on a conditional branch present after the spawning point—is the essence of SpMT. In Fig. 1d, thread `T2` is speculative because it is spawned from block B1 and the execution of `T2` is control dependent on the conditional branch in B3. Speculative spawning is particularly desirable when the length of the speculated path is long and the control flow is likely to take the speculated path more often than the other possible paths. As we will see later, speculative threads are a must for exploiting thread-level parallelism (TLP) from many nonnumeric programs. Nonnumeric programs also tend to have a noticeable number of control mispredictions, necessitating frequent recovery actions. Therefore, it is also important to exploit *control independence* [9], possibly by identifying threads that are nonspeculative from the control point of view. When executing such a nonspeculative thread in parallel with its spawning thread, a branch misprediction within the initiator does not affect the nonspeculative thread's existence (although it can potentially affect its execution through interthread data dependences). Effective use of control independence information thus helps to reach distant code, despite the presence of mispredicted branches in between. In Fig. 1d, thread `T3` is nonspeculative from its initiator (T1)'s point of view as it is spawned from B1 and the execution of `T3` is control-independent of the path taken to reach `T3` from B1.

2.3 Thread Granularity

Thread size is an important parameter to consider during program partitioning. Short threads may not expose adequate parallelism and may incur high overhead, depending on the thread initiation mechanisms. On the other hand, very long threads may be impractical due to expensive recovery actions from mispredictions and high buffering requirements.[1] In SpMT processors that organize the PEs as a circular queue [10], it is also important to reduce the variance in thread size [10].

2.4 Interthread Data Dependences

Another important factor to consider is interthread data dependences, which affect interthread data communication and determine the amount of TLP present. The impact of a data dependence depends on the producer's and consumer's respective positions in their threads. Detection of all data dependences at compile time is impossible because of aliasing. Accurate determination of the relative timing of the dependent instructions in different threads is also very difficult because of factors like conditional branches and cache misses. The compiler can, however, use profile information and heuristics to deal with this.

2.5 Prior Compiler Work on SpMT

The high level of complexity involved in program partitioning makes it a job more suitable for the compiler rather than the hardware. There have been several implementations of compiler-based thread generation for SpMT systems,

1. Even if a particular thread is nonspeculative from the control point of view, some of the data values used by that thread may be speculative. Because of this speculative nature, a thread cannot be committed until all of its data operands are verified to be correct and its results must be buffered until commit time.

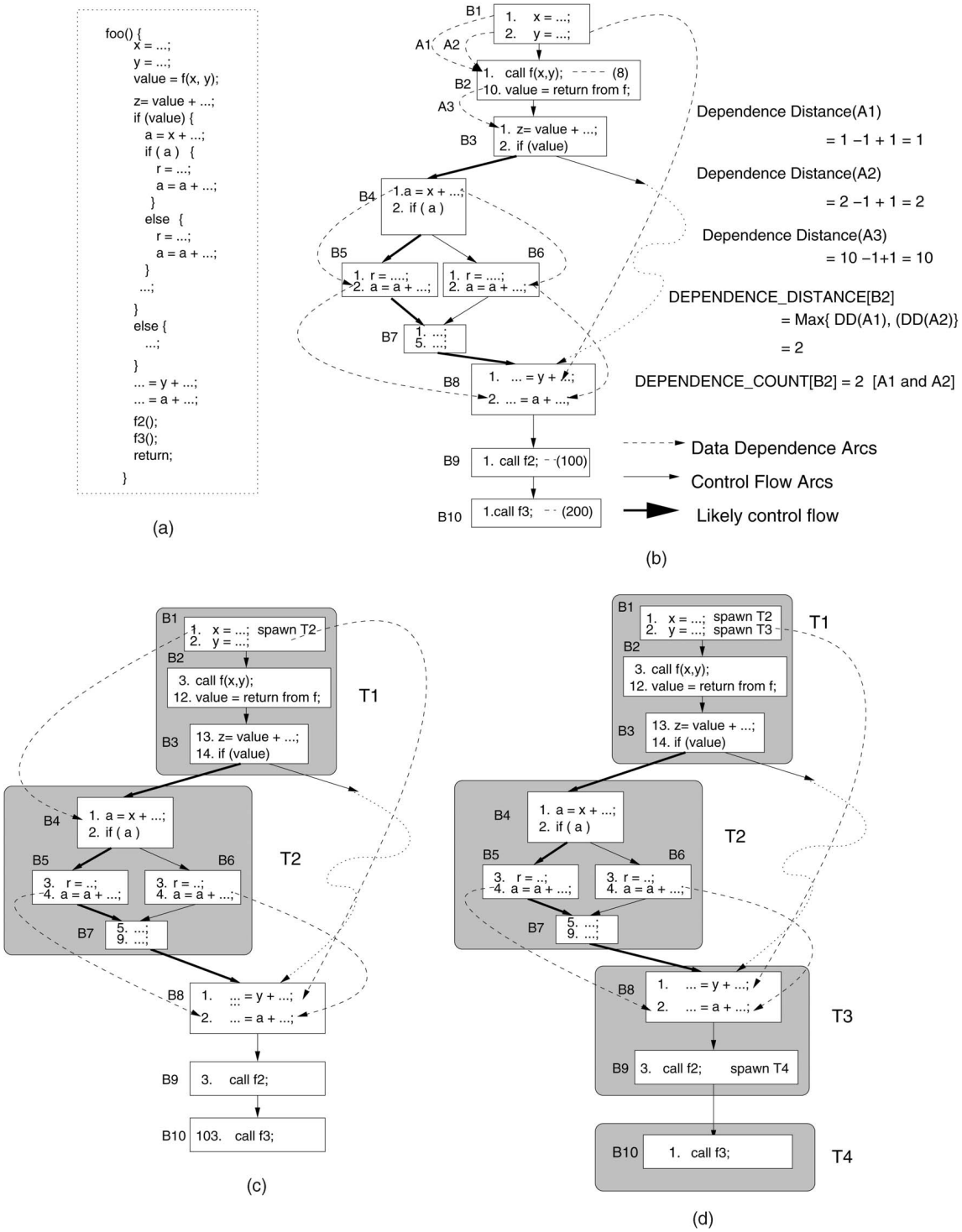

Fig. 1. Example of thread generation. (a) A sample C procedure—`foo()`. (b) The CFG for `foo()`. (c) After forming threads T1 and T2. (d) The threaded procedure `foo()`.

including superthreading [23], multiscalar [20], Hydra [16], clustered speculative multithreading [14], and Stampede [22]. Among these, the Agassiz compiler [23], the Hydra compiler [16], and the Stampede compiler [22] focus mainly on loop-centric threads. Agassiz also performs intrathread code scheduling for pipelined execution of threads. The Stampede compiler uses optimizations such as data forwarding and instruction scheduling to maximize the parallelism.

The multiscalar compiler [20] was the first major effort to partition the entire program. It uses a set of heuristics that are specific to the multiscalar model. It does not support out-of-order spawning of threads, in contrast to ours. For some programs, out-of-ordering spawning yields better performance, as shown in our experimental results. Furthermore, in the multiscalar model, a successor thread is spawned only from the beginning of a thread. Our compiler allows a thread to be spawned from anywhere within a thread.

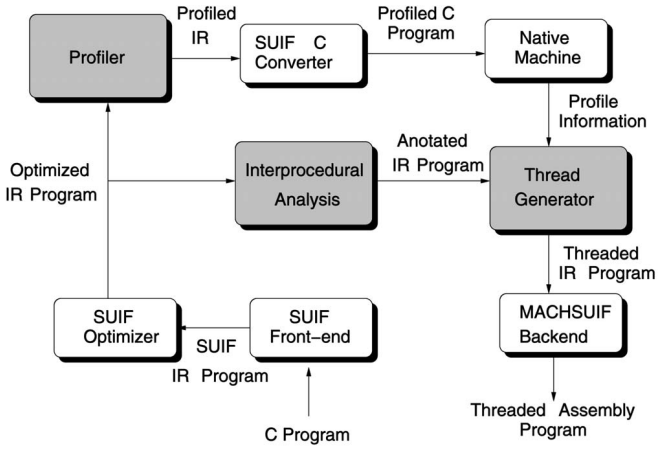

Fig. 2. The layout of our compiler framework for SpMT processors.

The spawning scheme proposed in [14] also partitions the entire program. It relies entirely on profiling to determine good spawning points and thread starting points. It computes *reaching probabilities* for all basic blocks, based on their execution frequency. Then, every pair of basic blocks is evaluated for likely pairs of {thread spawning point - thread starting point}, considering the likely thread size. This scheme does not take into account data dependences. On the other hand, our compiler considers statically determined data dependences besides profile-based information on likely control flow paths. Moreover, our compiler also uses value predictability information for obtaining better threads.

Apart from the SpMT model, there has been notable work on other highly speculative models such as *disjoint-eager execution (DEE)* [19] and *adaptive branch trees (ABT)* [8].

3 Our Compiler Framework

Next, we present our compiler framework for generating threads for speculative multithreaded processors. While partitioning a sequential program, our compiler considers *data dependences*, *control dependences*, and *thread size* together to decide a good partitioning. The compiler performs a thorough analysis of the program to generate efficient threads. We have developed some metrics and heuristics to help this endeavor. The layout of our compiler framework is shown in Fig. 2. It is built using the SUIF compiler framework [21] and has two main components: a *profiler* and a *thread generator*. The profiler collects execution statistics and the thread generator performs the partitioning.

The program to be partitioned is first converted into the SUIF intermediate representation (IR) using the SUIF front-end compiler *scc*. The IR program is then optimized by applying various optimizations available in SUIF, such as copy propagation, constant propagation, forward propagation, common subexpression elimination, dead code elimination, and moving loop-invariant calculation out of loops. The optimized code is passed through our interprocedural analysis module [5], [6], which annotates every instruction with the list of variables the instruction may read and write. The *read* and *write* sets of a call instruction contain the side effect of the procedure being called. The thread generator takes the annotated program, along with the profile information, and partitions it into threads. It also specifies the thread spawning points. Its output is in SUIF IR, from which the MACHSUIF [17] back-end generates the threaded Alpha assembly program.

Profiler: The compiler uses the profiler to determine the most likely paths in the program. The profile information is also used to estimate the number of dynamic instructions in a procedure (including the procedures called from that procedure), the size of a loop body, and the number of iterations of a loop.

3.1 Thread Generator

The thread generator partitions one procedure at a time. It does that in multiple passes. In the first pass, it builds the procedure's control flow graph (CFG). The immediate postdominator of every basic block is computed and the postdominator tree is built from the CFG. A main feature of our thread generator is explicit exploitation of control independences. The postdominator tree is used to find the control-independent point for each basic block. After building the CFG, the *use-def chains* [1] are computed using the *read* and *write* sets generated by interprocedural analysis. The interprocedural dependences are automatically taken care of by the *read* and *write* sets of the call instructions. Detailed array index analysis and structure analysis are done during the computation of *use-def sets*.

After computing the *use-def* information, the procedure is partitioned using Algorithm 1 (see Fig. 3). During partitioning, the effects of interthread data dependences are estimated by building an interthread data dependence model. Fig. 1b shows the CFG for procedure `foo()`. Note that while building a CFG, we use a new basic block for every call to a procedure. This facilitates our partitioning algorithm which partitions the program only at basic block boundaries. The likely path in the CFG from `B3` to `B8` is shown by thick arrows. Also, the average dynamic instruction count of each procedure is specified within parenthesis beside the procedure call.

3.2 Interthread Data Dependence Modeling

Interthread data dependences are very influential in program partitioning and should therefore be modeled as accurately as possible. This modeling is done on the fly during the partitioning process. We have incorporated two different models to quantify interthread data dependences. One uses *data dependence distance (DDD)* and the other uses *data dependence count (DDC)*.

3.2.1 Data Dependence Distance (DDD)

The *data dependence distance* between two threads `T1` and `T2` models the maximum time that the instructions in thread `T2` will stall for instructions in `T1` to complete, if `T1` and `T2` are executed in parallel. Consider the code segment in basic blocks {`B1, B2, B3`} in Fig. 1b. Instruction 1 of `B2` is data dependent on instructions 1 and 2 of `B1`, as shown by dependence arcs `A1` and `A2`. Assume sequential execution within each basic block and an instruction latency of one clock cycle. If `B1` and `B2` start execution in the same cycle, then instruction 1 of `B2` will wait one cycle for `x` and two cycles for `y`. So, in this example, B2 will encounter a delay of two cycles when executed in parallel with `B1`. Similarly, if `B2` and `B3` are executed in parallel, then the dependence distance of `B3` is 10 cycles because instruction 1 in `B3` has to wait for the return value of `f1()` in `B2`. On the other hand, if blocks `B4` and `B5` are executed in parallel, then instruction 2 of `B5` does not have to wait

Algorithm 1 Algorithm for program partitioning

```
partition_a_procedure (Procedure P)
 1: for all Loop L in P do
 2:    partition_loop(L);
 3: end for
 4: start_block = P.entry_block;
 5: pdom_block = postdominator block of start_block;
 6: curr_thread = create_new_thread (start_block, null);
 7: while (pdom_block != null) do
 8:    curr_thread = generate_thread (start_block, pdom_block, curr_thread);
 9:    start_block = pdom_block;
10:    pdom_block = postdominator block of start_block;
11: end while

Thread *generate_thread (start_b, end_b, curr_thread)
 1: pdom = postdominator block of start_b;
 2: path = most likely path between start_b and pdom;
 3: opt_dep = find_optimum_dependence (start_b, pdom, path, curr_thread, &spawn_instr);
 4: thread_size = sizeof (path) + sizeof (curr_thread);
 5: if (is_medium(thread_size) && (opt_dep < DEP_THRESHOLD)) then
 6:    add basic blocks from path to curr_thread;
 7:    curr_thread = create_new_thread (pdom, spawn_instr);
 8: else if (is_big (thread_size)) then
 9:    first_b = first basic blocks from path;
10:    opt_dep = find_optimum_dependence (start_b, first_b, null, curr_thread, &spawn_instr);
11:    if ( ! is_small(sizeof (curr_thread)) && (opt_dep < DEP_THRESHOLD)) then
12:       curr_thread_1 = create_new_thread (first_b, spawn_instr);
13:    else
14:       add first_b to curr_thread;
15:       curr_thread_1 = curr_thread;
16:    end if
17:    curr_thread_1 = generate_thread (first_b, pdom, curr_thread_1);
18:    opt_dep = find_optimum_dependence (start_b, pdom, null, curr_thread, &spawn_instr);
19:    if (opt_dep < DEP_THRESHOLD) then
20:       curr_thread = create_new_thread (pdom, spawn_instr);
21:    else
22:       curr_thread = curr_thread_1;
23:    end if
24: else
25:    add basic blocks from path to curr_thread;
26:    add pdom to curr_thread;
27: end if
28: curr_thread = generate_thread (pdom, end_b, curr_thread);
29: generate thread for other paths;
30: return curr_thread;
```

Fig. 3. Algorithm 1: Algorithm for program partitioning.

for the value of a from instruction 1 of B4 as it is already computed (assuming sequential execution). Formally,

$$\textbf{Dependence\_Distance}(\mathbf{T}) = MAX(dependence\_distance(A_i)); \forall \text{ incoming dependence edge } A_i \text{ of Thread } T.$$

The dependence distance for A_i is computed by considering the starting cycle of thread T relative to the predecessor threads, assuming enough processing elements.

It is not beneficial to parallely execute threads having large dependence distances. In order to decide whether to start a new thread at a certain point, our compiler calculates the dependence distance that will result if a new thread is started there. A thread is started only if the distance is below a threshold.

3.2.2 Data Dependence Count (DDC)

The *data dependence count* is the weighted count of the number of data dependence arcs coming into a thread from other threads. A small count value indicates that this thread is somewhat data independent on its predecessor threads, making it a good candidate to be executed in parallel with its predecessors. In Fig. 1b, the data dependence arcs for procedure foo() are shown with dashed arrows. If we consider blocks B1 and B2 as two separate threads, then the data dependence count of B2 is 2 because of the dependence arcs A1 and A2. While counting the dependence arcs, the compiler gives less weight to the arcs coming from distant threads as dependences from these threads are likely to be resolved earlier. Furthermore, the compiler gives less weight to the dependence arcs coming from the infrequent control flow paths. Formally,

$\mathbf{Dependence\_Count}(\mathbf{T}) = \Sigma w_n * A_n$;
where A_n is the number of dependence edges coming from thread T_n to T.

The weight w_n decreases (starting from 1) as the distance between T and T_n increases.

The rationale behind the use of data dependence count is twofold. First, it is simple to compute. Second, if out-of-order execution is carried out within a thread, then the data dependence distance model may not be very accurate because it assumes serial execution within each thread.

3.3 Program Partitioning

An overview of our partitioning approach is given in Algorithm 1 (see Fig. 3). The function `partition_a_procedure()` traverses the CFG and partitions the procedure into multiple threads. First, the loops are examined and partitioned in `partition_loop()`. Further partitioning is then done by traversing the CFG from the root. At every iteration of the *while loop* in `partition_a_procedure()`, the compiler looks ahead up to the control-independent point of the basic block under consideration and partitions the CFG between these two basic blocks into threads by calling `generate_thread()`. The function `create_new_thread()` starts a new thread from the basic block `start_block`, passed as the first argument. The second argument is an instruction (belonging to a predecessor thread) from where the new thread has to be spawned. The first thread in a procedure is actually the continuation of the thread containing the call to this procedure. Therefore, the second argument of `create_new_thread()` in line 6 of `partition_a_procedure()` is *null*. This function creates new entry into a global list of threads and returns a pointer to the newly created entry.

3.3.1 Partitioning Loops

Our framework treats loops as a special case of control dependences. It checks the data dependences between two successive iterations of a loop. If starting another thread at the next iteration is determined to be profitable, then a thread is started there. Notice that the spawning point can be kept somewhere inside the loop body and not necessarily at the beginning of the loop body. Large loops may be further partitioned into multiple threads, just like the case with nonloop regions.

Judicious partitioning of the loops is aided by profile information on the expected number of iterations and the number of dynamic instructions in the loop. In general, we do not want parallel execution of iterations of a small loop body that iterates only a few times. If a small loop body iterates many times, loop unrolling can first be done to increase the loop body size, after which the iterations can be executed in parallel. For partitioning nested loops, the compiler considers both the inner loop and the outer loop for parallel execution. Depending upon the available parallelism, the structure of the loop bodies, and the load balancing, either the inner loop, or the outer loop, or both can be designated as threads.

3.3.2 Thread Generation for Nonloop Regions

The pseudocode for `generate_thread()` is also shown in Algorithm 1 (see Fig. 3). This function takes two basic blocks and *current thread* as inputs and partitions the program segment between these two basic blocks into multiple threads (if possible) by calling itself recursively. *Current thread* consists of basic blocks from previous control-independent regions up to `start_block`. It first determines the most likely path between `start_block` and its immediate postdominator block (which is also the next control-independent point) based on the profile information. Then, it calls `find_optimum_dependence()` to determine the optimum dependence between *current thread* and the possible future thread starting from the postdominator. It also identifies the best spawning point for the future thread. Whether a thread gets spawned speculatively or nonspeculatively is determined by its spawning point. It is generally desirable to spawn a thread as early as possible. However, this also increases its data dependence on its predecessor threads. The function `find_optimum_dependence()` tries to find an early spawning point for the future thread while keeping its dependence distance/count below `DEPENDENCE_THRESHOLD`. The details of `find_optimum_dependence()` are available in [6].

To reduce thread size variance, our SpMT compiler maintains a lower limit and an upper limit for the number of dynamic instructions in a thread. If the current thread size (including the size of the likely path from the current basic block to the postdominator block) is within those limits and the dependence of the possible thread starting from the control-independent block is less than `DEPENDENCE_THRESHOLD`, then a new thread is started from the postdominator block. If the current thread is large, then the region between `start_block` and its control-independent block is further examined for possible partitioning. Moreover, if the possible thread starting from the control-independent point does not have much data dependence with the current thread, then a thread is created at the control-independent block. On the other hand, if the thread is too small even after including the blocks from the likely path, then no new thread is created at the control-independent point. The main goal of partitioning is to optimize the execution along the most likely path inside the procedure. The compiler also checks the less likely paths and partitions them as well.

3.3.3 Handling of Procedure Calls

The function `generate_thread()` automatically handles the call instructions present in the CFG. The compiler terminates the basic block after a procedure call. So, the instructions following a call instruction appear in the postdominator block of the basic block containing the call. At procedure call, it takes into account the average number of dynamic instructions to complete this call. The data dependences across the call are taken care of by the interprocedural analysis. If the called procedure is small, then it is completely included in the current thread. On the other hand, for a call to a bigger procedure, a new thread may begin after the call, depending on the thread size and the data dependences. If the called procedure is further partitioned into threads, then out-of-order spawning may take place.

3.4 An Example of Program Partitioning

The working of `generate_thread()` is illustrated with an example in Figs. 1c and 1d. In this example, we assume

that a thread should have more than eight instructions, but preferably less than 20 instructions.[2] We are using the data dependence distance model with a DEPENDENCE_THRESHOLD of 2, i.e., an instruction should not stall for more than two cycles for its operands from the predecessor threads, assuming sequential execution and one cycle latency. In Figs. 1c and 1d, we number the instructions with respect to the beginning of the threads to which they belong.

Program partitioning starts from block B1; the instructions of B1 are included in T1. Looking into B2, as T1's size at this point is less than 8, B2 is added to T1. The procedure f1() called from B2 has only eight dynamic instructions and, hence, is not partitioned into threads. Thus, the entire dynamic instance of f() is included in T1 and the size of T1 becomes 14. If we consider the thread size only, then we can start a new thread from B3 (as T1 now has more than eight instructions), but in that case, the dependence distance of B3 would be 12. Hence, we include B3 also in T1.

Afterward, the compiler determines the most likely path from B3 to its control independent block, B8 (shown by the thick arrows). If we include the blocks {B4, B5, B7} in T1, then T1's size becomes more than 20. Therefore, we try to start a new thread T2 from B4 by finding a suitable spawning point from T1. The function find_optimum_distance() in line 10 of the pseudocode of generate_thread() does that. It first builds a possible future thread T2 containing {B4, B5, B7}. After that, it finds a spawning point for T2 by computing the dependence distances of T2 for various spawning points. We see that, if T2 is spawned from instruction 1 of T1, the resultant dependence distance for T2 is 1, i.e., less than DEPENDENCE_THRESHOLD. Therefore, T2 is started from B4 and spawned speculatively from instruction 1 of T1, as shown in Fig. 1c. Then, we include blocks {B5, B6, B7} in T2. If that path were longer, it would have been partitioned into multiple threads.

After partitioning the most likely path from B3 to B8, we recompute the optimum dependence for a future thread T3 starting at B8, considering both T1 and T2. In Fig. 1, we see that if T3 is spawned from instruction 1 of T1, then the dependence distance is 3 due to the dependence arc from instruction 4 of T2 to instruction 2 of T3. However, if we spawn T3 from instruction 2 of T1, then the dependence distance is 2, which is within DEPENDENCE_THRESHOLD. Hence, we spawn the nonspeculative thread T3 from there, as shown in Fig. 1d. After starting T3 from B8, we look into B9. Like B2 in T1, we include B9 in T3. The number of dynamic instructions in f2() called from B9 is 100 and f2() is therefore partitioned into threads. Moreover, by performing interprocedural analysis, the compiler finds that there is no dependence between f2() and f3(). Therefore, it spawns thread T4, starting at the call of f3() from T3. Note that T4 is spawned out-of-order as it is spawned before the threads belonging to f2(), which precede T4 in program order.

4 Exploiting Value Predictability in Thread Generation

Data dependences have a big effect on the speedup obtained in a multithreaded system. If interthread data dependences abound, very little speedup will result unless the SpMT hardware uses data value prediction. Past research has shown that runtime data value prediction can provide good speedup in SpMT processors by temporarily ignoring data dependences and executing instructions based upon the predicted values [15]. When employing data value prediction, the partitioning algorithm could be improved to exploit this. We next present an enhancement that considers the effects of runtime data value prediction while making compile-time partitioning decisions. To do this, we map the predictable instructions to the actual source variables they operate on and then use the predictability information of the source variables to build a more accurate data dependence model.

2. Note that, because of the library routine calls and program structure, it is not always possible to restrict the thread size within the upper limit.

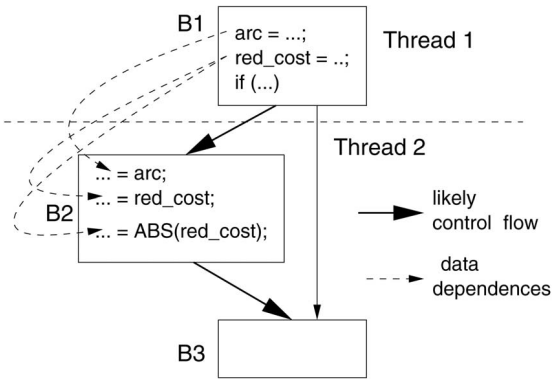

Fig. 4. An example code fragment to show the importance of data value profiling.

4.1 Importance of Exploiting Value Predictability Information

We shall use an example to illustrate the importance of exploiting value predictability information at compile time. The code fragment (and CFG) shown in Fig. 4 is taken from a frequently executed function in mcf, a SPEC2000 program. Most of the time the control goes to B2 after executing B1. Under the partitioning scheme described earlier, a new thread is not created at the beginning of B2 for parallel execution with B1 because there are many data dependences between B1 and B2 through variables *arc* and *red_cost*. So, B1 and B2 are grouped under the same thread.

However, the data value prediction statistics indicate that the variables *arc* and *red_cost* are correctly predicted more than 80 percent and 52 percent of the time, respectively. Therefore, at runtime, more than 50 percent of the time, it is actually beneficial to execute B1 and B2 in parallel. By using the prediction statistics of these variables, the compiler can ignore the related data dependences and allocates these basic blocks into two separate threads.

4.2 Integrating the Effects of Data Value Predictability

The first step toward considering the effects of runtime value predictability during the compilation phase is to collect runtime prediction statistics. We developed a profiler called *value predictability profiler (VPP)* for this purpose. For every variable accessed during profiling, VPP determines the number of times it was accessed and the number of times it was predicted correctly. The challenge is that the VPP works with the executable, which has no notion of progam variables, whereas the thread generator needs the information for the program variables. Therefore, all information collected by VPP needs to be mapped back to the IR code.

VPP executes the program using a trace-driven simulator, which has a data value predictor similar to the one to

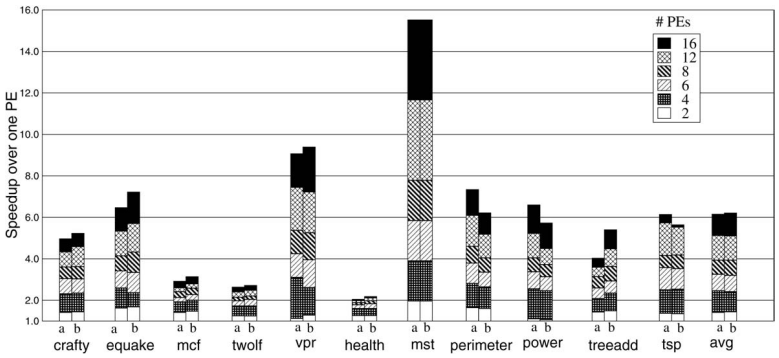

Fig. 5. Speedup obtained with different dependence modeling. Bar a: Data dependence distance. Bar b: Data dependence count.

be used in the SpMT processor. VPP collects the prediction statistics for instructions as and when they are simulated and simultaneously maps their source operands to the corresponding source variables, if possible, using the symbol table of the executable. Not all operands correspond to source variables; often, instructions either access registers that hold intermediate results or load register-spilled values from memory locations. A global variable is identified by matching the memory address with the elements of the symbol table. Local variables are identified by determining their position in the current stack frame. For structure variables, VPP maps the source operand to the particular field of the structure it corresponds to. For array variables, VPP maps the source operand to the array variable and does not narrow it down further to the specific index. However, our SpMT compiler later uses the prediction statistics of the array variable along with array index analysis to determine whether the dependence due to the array variable can be ignored.

Using the value prediction statistics in the thread generator is quite straightforward: In the interthread dependence model, consider only the dependences due to the variables with value prediction accuracies less than a threshold.

5 Experimental Evaluation

Our simulator models a generic SpMT processor on top of a trace-driven simulator. Each processing element (PE) has its own program counter, fetch unit, decode unit, and execution unit that can fetch and execute instructions from a thread. Each PE has an instruction window of 64 instructions and can issue up to four instructions per cycle in an out-of-order fashion. The PEs are connected together by a 2-cycle latency interconnect. All functional units are assumed to have a single cycle latency. There is a 2 cycle overhead in assigning a thread to a PE; thread preemption also incurs a 2 cycle penalty. The simulator also models a shared 256 KB L1 d-cache with a 1 cycle access latency and a miss latency of 10 cycles; we have kept the memory system somewhat ideal so that the results are not tied to a specific memory system. When encountering a conditional branch instruction in a thread, a branch predictor is consulted for making a prediction. A hybrid data value predictor is modeled for predicting the results of instructions whose operands are unavailable at decode time.

The simulator uses the Alpha ISA. The code executed in the supervisor mode is unavailable to the simulator and are therefore not taken into account in the statistics. The library code is not parallelized as we use the standard libraries in our experiments. The serial execution of the library code provides a conservative treatment to our parallelism values.

For benchmarks, we use 11 programs: five from SPEC2000 and six from the Olden suite, all written in C. For the SPEC benchmarks, we "fast forward" the first 500 million instructions in order to skip the initialization phases before beginning the actual simulation. For the Olden benchmarks, there is no need to fast forward as they do not have any initialization phases. We simulate all the programs except `mcf` and `perimeter` for one billion instructions after fast forwarding (the parallelism values are found to be stable at one billion instructions); `mcf` terminated after 700 million instructions, while `perimeter` terminated after 500 million instructions. For the SPEC benchmark programs, we use the *train* data set as input during profiling and the *ref* data set as input during simulation (except for `mcf`). For `mcf`, we use the `lgred` input from the MinneSPEC [11] input set, as its `ref` input has very high memory requirements. In Olden benchmarks, we use the same input sizes as [7] for the simulation and scaled down input for profiling.

Default Partitioning Setup: As there are many parameters, it is difficult to perform a completely orthogonal set of experiments. Therefore, we define a default setup, and vary one parameter at a time. For the default setup, the compiler generates all kinds of threads (i.e., *speculative* threads, *control independent* threads, and *loop-centric* threads) and uses data dependence distance-based modeling of interthread data dependences. The default compiler threshold values for lower and upper limits of the number of dynamic instructions in a thread are 20 and 200, respectively. The default value for interthread data dependence distance threshold is 15.

5.1 Evaluation of Basic Thread Generation Scheme

To evaluate the effectiveness of our basic partitioning algorithm, we measure the speedup obtained with all kinds of threads by increasing the number of PEs from 1 to 16. Fig. 5 shows the speedup measured with respect to the sequential execution of the programs in a single PE. Table 1 presents some thread-related statistics for the DDD-based configuration used in Fig. 5.

On analyzing the speedup obtained with the DDD-based partitioning, we see that most of the benchmarks show good speedup and scalability as we increase the number of PEs. From Table 1, we see that more than 83 percent of the dynamic instructions in `crafty` belong to nonloop threads and the fact that it shows good speedup and scalability

TABLE 1
Runtime Thread Statistics

Prog. Name	Avg. Dynamic Thread Size	% of Dynamic Instr. of Diff. Thread Types		
		Speculative	Nonspeculative	Loop-centric
crafty	93.5	63.8%	14.1%	22.1%
equake	31.4	1.6%	0.3%	98.1%
mcf	35.7	3.5%	0.1%	96.4%
twolf	33.5	13.1%	8.3%	78.6%
vpr	80.1	34.3%	16.4%	49.3%
health	8.9	5.6%	0.0%	94.4%
mst	1146.4	0.0%	0.0%	100.0%
perimeter	67.7	78.6%	21.4%	0.0%
power	42.6	4.5%	85.0%	10.5%
treeadd	106.5	100.0%	0.0%	0.0%
tsp	103.8	16.1%	0.1%	83.8%

indicates that our compiler has been successful in extracting parallelism from the *nonloop* parts of the code.[3] This is true for `vpr`, `perimeter`, `power`, and `treeadd`. Benchmarks `perimeter` and `treeadd` do not have loops; instead, they have recursive function calls. These benchmarks have large percentages of *speculative* threads and *nonspeculative* threads.

In {`equake`, `mcf`, `twolf`, `health`, `mst`, `tsp`}, most of the time is spent in loops. For `mst`, the speedup is very high, linear in the number of PEs, as it spends most of the time in a parallelized loop with no loop-carried dependences. `mcf` and `health` suffer mainly from too many cache misses. `twolf` and `health` have very little loop-level parallelism.

In Table 1, we see that the average thread size is reasonable for all programs except `mst` and `health`. In `health`, there is a small loop body that is executed most of the time, resulting in small threads and, therefore, the PEs are not able to exploit TLP well. This loop mainly traverses a linked list and, therefore, prefetching is required to get parallelism from this loop. On the other hand, in `mst`, the most frequently executed *loop-centric* thread contains library routine calls that our compiler did not partition, resulting in very large threads.

5.2 Comparison between the Two Data Dependence Models

From Fig. 5, we can also compare the effectiveness of the two data dependence models used for thread generation. These results are a mixed bag. The Olden benchmark programs `treeadd` and `perimeter` show the most significant differences in performance. For `perimeter`, DDD-based modeling gives a higher speedup, while, for `treeadd`, DDC-based modeling gives a better speedup.

On analyzing `treeadd`'s threads, we found that the same set of threads are generated in both cases, but the spawning points are different. One of the most executed threads is spawned from an earlier point in the DDD-based partitioning than in the DDC based partitioning. The DDD model estimated that a particular dependence would get resolved early at runtime and spawned the thread accordingly. But, it did not get resolved early at runtime and the thread's PE stalled for that data affecting the overall speedup. In general, both models seem to be quite effective in modeling interthread data dependences.

3. In this context, the term *loops* excludes those loops whose bodies contain procedure calls such that successive iterations of the loops are thousands to millions of instructions apart, e.g., the processing loop in the *main()* function.

5.3 Effectiveness of Different Types of Threads

To evaluate the contribution of different types of threads on performance, we conducted experiments by varying the nature of threads. For this study, DDD-based modeling was used. Fig. 6 presents the results obtained with three different combinations of thread types. From this figure, we can see that loop-centric threads alone are quite insufficient to harness the parallelism present in most of the benchmarks.

From Table 1, we see that more than 65 percent of the threads in `crafty` are *speculative*, so, *loop-centric* threads alone or together with *nonspeculative* threads could not completely exploit the available parallelism. In `equake`, although only 5 percent of the threads are either speculative or nonspeculative, they have a significant contribution to parallelism. We found that, by depending solely on loop-based threads, some parts of the program become completely serialized, which affects the overall speedup of the program. In `tsp`, although only around 15 percent of the threads are *speculative*, they seem to play a key role in exploiting parallelism. It may be possible that, by not spawning the speculative threads, load balancing and thread scheduling get affected, thereby affecting the performance.

5.4 Effectiveness of Accounting for Data Value Predictability

The results presented in Fig. 7 show the impact of using value predictability information in thread formation with a dependence threshold of 50 percent. Table 2 shows the overall prediction statistics of the source variables for the benchmark programs.

Comparing bar *a* with *b* and *c* with *d* in Fig. 7, we find that `mcf` shows a 39 percent increase in speedup for DDD-based modeling and 32 percent increase for DDC-based modeling, when using 16 PEs. The reason behind this large improvement was already pointed out in Section 4. `crafty`, `vpr`, and `perimeter` show significant improvement in speedup when accounting data value predictability. From these results we can infer that some of the variables that play a crucial role in deciding the partitioning in these benchmark programs have good prediction accuracies. By ignoring the dependences due to these variables, our compiler has performed a better partitioning.

The benchmarks `equake`, `mst`, and `power` do not show additional speedup, although they have very high prediction accuracies. In `mst`, all of the parallelism is present in a single loop, and the compiler could extract it without accounting for data value prediction. For `equake` and `power`, it appears that the variables that have high prediction accuracies are not playing a critical role in thread formation. For example, if a variable is written and then read inside the same basic block, then this data dependence is not considered for deciding the partitioning. Similarly, the dependences coming from distant threads also do not have a major influence on the partitioning. From Fig. 7, we see that, for most of the benchmarks, the

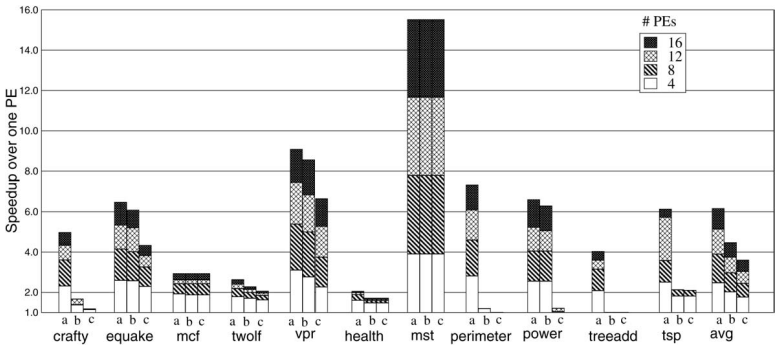

Fig. 6. Speedup obtained with different types of threads. Bar a: Speculative + nonspeculative + loop-centric threads. Bar b: Nonspeculative + loop-centric threads. Bar c: Loop-centric threads.

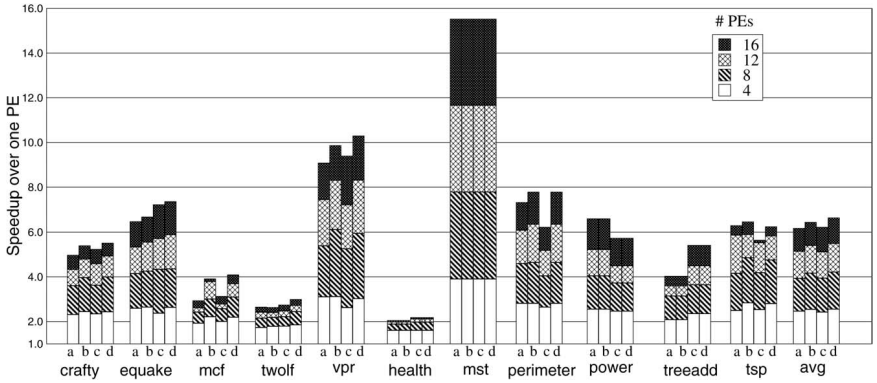

Fig. 7. Speedup obtained with threads formed by different partitioning schemes. Bar a: Data Dependence Distance modeling without using value predictability. Bar b: Data Dependence Distance modeling with using value predictability. Bar c: Data Dependence Count modeling without using value predictability. Bar d: Data Dependence Count modeling with using value predictability.

difference in speedups for the two dependence models becomes less significant when utilizing value predictability information.

5.5 Effectiveness of Out-of-Order Spawning

Our next set of experiments focus on the effect of out-of-order thread spawning. The results are shown in Fig. 8, where the bars are marked with their out-of-order spawning depth. Our compiler framework can create out-of-order spawning to an infinite depth, but it is not practical for the SpMT hardware to support infinite depth of out-of-order spawning because of limited buffer space within each PE.

TABLE 2
Prediction Accuracies of the Source Variables

Program Name	No. of Variables		Overall Prediction Accuracies of Variables
	Accessed	with Prediction Accuracy $\geq 50\%$	
crafty	1088	176	34.19%
equake	172	51	75.49%
mcf	156	54	51.62%
twolf	1727	156	41.34%
vpr	511	124	48.00%
health	55	11	4.10%
mst	64	28	88.49%
perimeter	22	9	39.76%
power	85	21	99.76%
treeadd	14	0	11.65%
tsp	65	4	24.46%

Also, in order to support out-of-order spawning, the SpMT processor may have to frequently preempt some of the (sequentially older) threads, thereby increasing the overhead. So, ideally, we would like to extract as much parallelism as possible without any out-of-order spawning or at a low out-of-order spawning depth.

In Fig. 8, we see that, except for `health` and `mst`, all other benchmarks show marked speedup improvements even at at an out-of-order spawning depth of 2. This implies that, in `health` and `mst`, even in the default configuration, the threads are mostly spawned and executed in sequential order. In `tsp`, there is an increase in parallelism only when the depth is increased from 4 to ∞. This signifies that there is some parallelism available at a distance. In `power`, there is a 139 percent increase in speedup as the depth is increased from 0 to 2. In `power`, the most frequently executed part of the program contains calls to procedures that can be executed in parallel. The first procedure called is again partitioned into two threads. Without out-of-order spawning, the procedures are sequentially executed. However, by allowing an out-of-order spawning depth of 1, the second procedure can be executed in parallel with the first procedure, resulting in significant speedup.

5.6 Impact of Thread Size and Data Dependence Thresholds

To evaluate the impact of the thread size threshold and the data dependence threshold values used by the compiler, we compiled the benchmarks programs with different sets of threshold values and measured the performance. The thread size threshold is defined by two threshold values: the lower limit and the upper limit of the number of

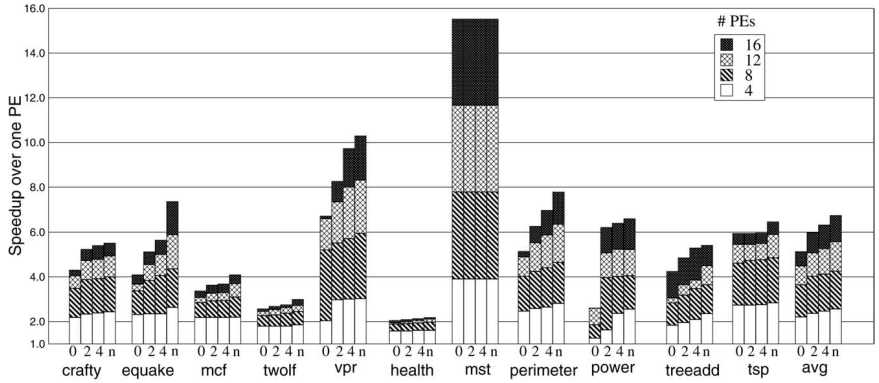

Fig. 8. Speedup obtained by restricting out-of-order spawning depth to 0, 2, 4, and ∞ (denoted by n).

dynamic instructions in a thread. The compiler tries to create threads within the specified size limits. The amount of allowable interthread data dependence is controlled by the data dependence threashold.

We chose threshold values with significant differences to observe their impact. In Fig. 9, we present the results for the SPEC benchmarks, for varying thread size thresholds. From the figure, we see that the highest speedups are obtained when the lower and upper limits are 20 and 200, respectively (i.e., bar "c"), that are the default values for all other experiments. Within this limit, the processors are able to extract parallelism at different granularities. Moreover, it could also exploit ILP within each thread.

The lower and upper limits of 8 and 40 (i.e., bar "a") generate smaller size threads that do not capture parallelism at larger granularities. Moreover, small threads have less ILP and the thread starting overhead may offset the advantage of exploiting TLP. The significant decrease in speedup for `crafty` and `vpr` in bar "b" could be because of the high thread size variance and lack of parallelism at higher granularity.

Fig. 10 shows the speedup obtained by varying the data dependence distance threshold to 5, 15 (default value), and 50. For lower dependence thresholds, the compiler partitions more conservatively and, therefore, it loses parallelism opportunities available at runtime due to data value prediction and out-of-order execution. By comparing the speedup with the threshold values of 15 and 50, we see that speedup increases for `equake` and `mcf` while it decreases for the other benchmarks. The data value prediction accuracies of `equake` and `mcf` are quite high; by using a higher threshold, the compiler could effectively ignore the dependences due to the predictable variables resulting in a higher speedup. For other benchmarks, the use of higher dependence threshold results in threads that are stalled more often because of interthread data dependences and, thereby, reducing the speedup.

6 Conclusions

Speculative multithreading (SpMT) is emerging as an important parallelization method for nonnumeric programs. The main idea is to fetch and execute multiple threads, derived from a single program, in parallel. Judicious partitioning of a sequential program into threads is difficult to do in hardware. Previous compiler efforts have focused mainly on loop-based threads and speculative threads. A limitation of this is that branch mispredictions may cause all subsequent threads to be squashed, without retaining any active nonspeculative threads. The use of nonspeculative threads has the potential to extract additional amounts of parallelism.

This paper presented a general compiler framework for partitioning a sequential program into multiple threads for execution in an SpMT processor. This compiler framework has been implemented on the SUIF-MachSUIF platform. Our compiler framework is general and can identify loop-based threads, speculative threads, and nonspeculative threads. In

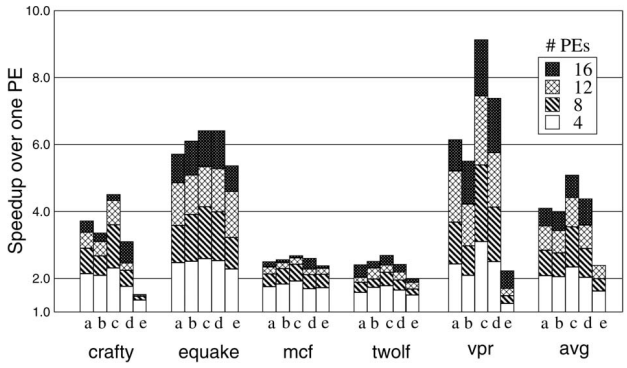

Fig. 9. Speedup obtained with different thread size limits. Bar a: 8 (lower)-40 (upper). Bar b: 8-200. Bar c: 20-200. Bar d: 20-500. Bar e: 100-500.

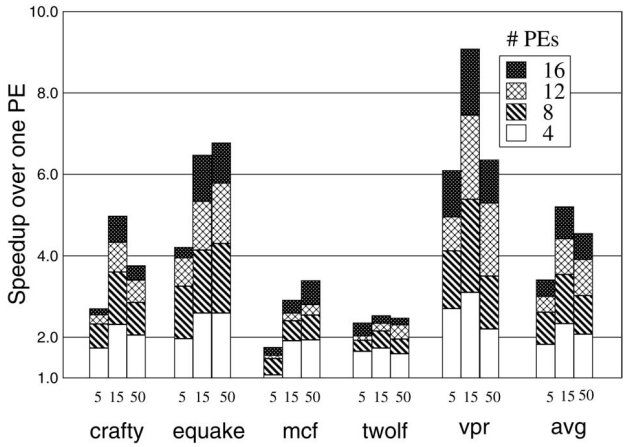

Fig. 10. Speedup obtained with different dependence distance thresholds (values shown under the bars).

addition, it also supports out-of-order spawning of threads, and spawning from anywhere in a thread. While performing the program partitioning, the compiler not only considers control independence information, but also considers data dependence information and profile-based information on the most likely control flow paths. Our compiler framework is therefore useful for a variety of SpMT architectures. A simulation-based evaluation of the threads generated with our compiler indicates that the combination of loop-based threads, speculative threads, and nonspeculative threads has the potential to extract large amounts of thread-level parallelism from nonnumeric programs.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation (through grants CCR 0073582 and CCR 9988256), Intel Corporation, and IBM Research.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
[2] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proc. Int'l Symp. Microarchitecture,* 1998.
[3] A. Bhowmik and M. Franklin, "A General Compiler Framework for Speculative Multithreading," *Proc. 14th Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 99-108, 2002.
[4] A. Bhowmik and M. Franklin, "Exploiting Data Value Prediction in Compiler Based Thread Formation," *Proc. Int'l Conf. High Performance Computing,* 2002.
[5] A. Bhowmik and M. Franklin, "A Fast Approximate Interprocedural Analysis for Speculative Multithreading Compilers," *Proc. Int'l Conf. Supercomputing,* 2003.
[6] A. Bhowmik, "A General Compiler Framework for Speculative Multithreaded Processors," PhD thesis, Computer Science Dept., Univ. of Maryland, 2003.
[7] M.C. Carlisle and A. Rogers, "Software Caching and Computation Migration in Olden," *Proc. Symp. Principles and Practice of Parallel Programming,* 1995.
[8] T.-F. Chen, "Supporting Highly Speculative Execution via Adaptive Branch Trees," *Proc. Int'l Symp. High-Performance Computer Architecture,* 1998.
[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages and Systems,* vol. 13, Oct. 1991.
[10] M. Franklin, *Multiscalar Processors.* Kluwer Academic, 2002.
[11] A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters,* vol. 1, June 2002.
[12] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers,* vol. 48, no. 9, Sept. 1999.
[13] P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," *Proc. Int'l Conf. Supercomputing,* 2001.
[14] P. Marcuello and A. Gonzalez, "Thread Spawning Schemes for Speculative Multithreading," *Proc. Int'l Symp. High-Performance Computer Architecture,* 2002.
[15] P. Marcuello, J. Tubella, and A. Gonzalez, "Value Prediction for Speculative Multithreaded Architectures," *Proc. Int'l Symp. Microarchitecture,* 1998.
[16] K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," *Proc. Int'l Conf. Supercomputing,* 1999.
[17] M.D. Smith and G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization," http://www.eecs.harvard.edu/hube/software/nci/overview.html, 2004.
[18] J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques,* 1996.
[19] A. Uht, V. Sindagi, and K. Hall, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. Int'l Symp. Microarchitecture,* 1995.
[20] T.N. Vijaykumar and G.S. Sohi, "Task Selection for a Multiscalar Processor," *Proc. Int'l Symp. Microarchitecture,* 1998.
[21] R. Wilson et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compiler," *ACM SIGPLAN Notices,* vol. 29, no. 12, Dec. 1996.
[22] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, "Compiler Optimization of Scalar Value Communication between Speculative Threads," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2002.
[23] B. Zheng, J.-Y. Tsai, B.Y. Zhang, T. Chen, B. Huang, J.H. Li, Y.H. Ding, J. Liang, Y. Zhen, P.-C. Yew, and C.Q. Zhu, "Designing the Agassiz Compiler for Concurrent Multithreaded Architecture," *Proc. Workshop Languages and Compilers for Parallel Computing (LCPC),* 1999.

Anasua Bhowmik received the PhD degree in computer science from the University of Maryland, College Park, in 2003. She received the ME (1997) degree in computer science and automation from the Indian Institute of Science and the BE (1995) degree in computer science and engineering from Jadavpur University, India. She is a research associate in the Computer Science and Automation Department at the Indian Institute of Science, Bangalore. Her research interests include compiler and architecture for high performance computing, compilation techniques for embedded systems, and programming languages.

Manoj Franklin received the BSc (Engg) degree in electronics and communications engineering from the University of Kerala and the MS and PhD degrees in computer sciences from the University of Wisconsin-Madison. He is an associate professor in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. He has a joint appointment with the University of Maryland Institute of Advanced Computer Studies (UMIACS), and an affiliate appointment with the Department of Computer Science at the University of Maryland. Prior to joining the University of Maryland, he had worked as a faculty member in the Department of Electrical and Computer Engineering at Clemson University, South Carolina. Professor Franklin's research interests are in computer architecture and compilers. His research is funded by the US National Science Foundation, Intel Corporation, and IBM Research.