A General Compiler Framework for Speculative Multithreading

Anasua Bhowmik
Computer Sciences Department
University of Maryland
College Park, MD 20742
anasua@cs.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

ABSTRACT

Speculative multithreading (SpMT) promises to be an effective mechanism for parallelizing non-numeric programs, which tend to use irregular data structures with pointers and have complex flows of control. Proper thread selection is crucial to 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 support a wide variety of threads, such as speculative threads, non-speculative threads, loop-centric threads, and out-of-order thread spawning. To do efficient partitioning, the compiler uses profiling, intra-procedural pointer analysis, data dependence information and control dependence information. Our compiler framework is implemented on the SUIF-MachSUIF platform, and is able to partition large programs, such as the SPEC benchmarks. A simulation-based evaluation of the generated threads shows that an average speedup of 3 can be obtained with 6 processing elements for non-numeric programs. This speedup reduces to 2 if we use only loop-based threads.

Keywords: data dependence, parallelization, speculative multithreading (SpMT), thread-level parallelism (TLP)

1. INTRODUCTION

Reducing the completion time of a single computation task has been one of the defining challenges of computer science and engineering for the last several decades. The primary means of increasing processor performance, besides increasing the clock speed and reducing the memory latency, has always been the exploitation of the inherent parallelism present in programs, with the use of a combination of software and hardware techniques. Parallelization has been a good success for scientific applications, but not quite so for the non-numeric application. Non-numeric programs use irregular data structures and have complex control flows that make them hard to parallelize.

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.

The emergence of the *speculative multithreading* (SpMT) model in the last decade has provided the much awaited breakthrough for the important set of non-numeric applications. Many studies on speculative multithreading (SpMT) confirm that there is significant performance potential in executing multiple threads from the same program in parallel.

Hardware support for speculative thread execution makes it possible for the compiler to parallelize sequential applications without worrying about data and control dependences. However, hardware support for speculation is not sufficient to achieve high speedup from the application programs and we need good compiler support as well to extract parallelism from the programs. In compiling programs for the multithreaded architecture the most important task is thread partitioning, i.e., partitioning a program into separate threads of execution.

The major contribution of this paper is to present and evaluate a general compiler framework for SpMT systems. This compiler partitions sequential programs into multiple threads for parallel execution in an SpMT processor. Our focus is primarily on non-numeric applications, which are generally more difficult to partition into threads. Traditional work in parallelization has targeted scientific applications, and has focussed mainly on loops where the loop bounds are generally predefined and the loops access regular data structures like arrays. On the other hand, in non-numeric applications, the loops often have large loop bodies with complex control flow, loop-carried dependences and loop bounds that cannot be resolved statically. So these loops cannot be easily parallelized with traditional techniques. Also, unlike scientific applications, non-numeric applications access irregular data structures with an abundance of pointers. Moreover, sometimes the non-numeric programs spend more time outside the loops. So many of the techniques used for scientific programs cannot be directly applied to non-numeric programs to extract parallelism.

To obtain good speedup for non-numeric programs, our compiler considers both the loop regions and the *non-loop* regions of programs. It uses control dependence information and profile information to guide the partitioning. We have used SUIF and MACHSUIF compiler platforms to develop our compiler. Using our compiler framework, we have been able to compile a wide range of non-numeric applications, including programs from the SPEC 2000 and Olden benchmark suites.

Our work differs from earlier works on SpMT compilation [13] [18] [19] primarily in 4 ways: (i) Most of the earlier work [13] [18] primarily targets loop-level parallelism only, whereas our compiler targets other kinds of parallelism also. (ii) Our SpMT model is more general than the one used in earlier compiler work, and supports spawning of threads from anywhere in a thread; in [19] a thread can be spawned only from the beginning of another thread. (iii) Our compiler framework supports out-of-order spawning of threads, whereas earlier compilers support only sequential spawning of threads. (iv) Our Compiler framework explicitly exploits control dependence information in forming the threads.

Our studies with different types of compiler-generated threads have led to the following conclusions:

- Significant speedups can be obtained with low degrees of multithreading for the non-numeric applications.
- For non-numeric programs, it is *not* sufficient to exploit loop-level parallelism only, the form of parallelism that is almost exclusively targeted in prior research; it is important to look at other types of threads as well.
- For non-numeric programs, it is important to spawn threads speculatively.
- For non-numeric programs, it is important to exploit control independence.

The rest of this paper is organized as follows. Section 2 provides background information on SpMT, including the thread execution model and various issues related to thread partitioning. Section 3 details our SpMT compiler framework, and thread partitioning algorithm. Section 4 presents the simulation environment and a detailed evaluation of our thread partitioning algorithm. Section 5 presents a summary and the major conclusions of this paper.

2. SPECULATIVE MULTITHREADING (SPMT)

Compilers and programmers have made significant progress in parallelizing regular numeric applications. However, they have had little or no success in doing the same for highly irregular numeric or especially non-numeric applications [9]. In such applications, control flow as well as memory addresses often depend on run-time behavior, which makes it very difficult to partition a program into independent threads.

This execution model is closer to sequential control flow, and envisions a strict sequential ordering among the threads. Threads are extracted from sequential code and are speculatively run in parallel, without violating the sequential program semantics. In case of misspeculation, the results of the speculative thread and of subsequent threads are discarded. The control flow of the sequential code imposes an order on the threads, we can use the terms *predecessor* and *successor* to qualify the relation between any given pair of threads. This means that inter-thread communication between any two threads (if any) is strictly in one direction, as dictated by the sequential thread ordering. Thus, no explicit synchronization operations are necessary, as the sequential semantics of the threads guarantee proper synchronization. This relaxation allows us to "parallelize" non-numeric applications without explicit synchronization, even if there is a potential inter-thread data dependence.

Example SpMT models are the multiscalar model [4] [16], the superthreading model [18], and the trace processing model [8] [14]. SpMT is appealing because it provides the power of parallel processing to speed up ordinary applications, which are typically written as sequential programs.

2.1 SpMT Thread Communication Model

Inter-thread communication refers to passing data values between two or more threads. Communication can take place at the level of register space, memory address space, and I/O space, with the registers being the level closest to the processor. The most general model, which is followed in most of the SpMT proposals, is to let inter-thread communication take place at all of these levels. Thus, multiple threads share the same register name space (and the same memory address space). Inter-thread communication happens implicitly due to reads and writes to the shared registers[1] (and to shared memory locations). Our compiler framework also uses this most general communication model.

2.2 Spawning Strategies

In an SpMT processor, a dynamic thread's lifetime has 3 important events: spawning, activation, and retirement. Spawning refers to creating a new instance of a static thread, and is analogous to the fork mechanism used in conventional parallel processing. Activation refers to assigning a spawned thread to a processing element (PE). Retirement refers to the act of a completed thread relinquishing its PE (after it has committed its results).

2.2.0.1 Spawning Point:.

An important issue in an SpMT model concerns the points in a thread from where other threads are spawned. Two possibilities exist:

- Spawning from only the beginning of a thread
- Spawning from anywhere in a thread

The first case uses an eager spawning strategy, with a view 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 run-time information. Furthermore, often there may not be an idle PE at the time a thread is spawned. In the second approach, a thread can be spawned from anywhere within a thread. This allows the spawning to be delayed, say, until a particular branch or data dependence gets resolved.

2.2.0.2 Loop Iterations versus Non-loop Threads:.

Loop iterations have been the traditional target of parallelization at all levels—programmer, compiler, and hardware—and form an obvious candidate for forming threads. Each iteration of a loop can be specified as a thread that runs in parallel with other iterations of that loop. For example, in Figure 1(b), *Thread 1* is a loop-centric thread, i.e every iteration of the loop is executed 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 towards loop continuation, even in non-numeric programs. The degree of TLP that can

[1] A shared register name space can be implemented at the microarchitecture level in a distributed manner.

be extracted will be moderated, however, by loop-carried dependences. In non-numeric programs, many of the loops have at least some amount of loop-carried data dependences. To get good speedup for non-numeric programs, it is important to consider threads other than loop iterations, in addition to loop iteration based threads.

2.2.0.3 Speculative versus Non-speculative Threads:.

Speculative spawning is the essence of SpMT architectures. A speculative spawning is where the existence of the spawned thread is control dependent on a conditional branch that follows the spawning point (as per sequential program order). Many non-numeric programs, however, tend to have a noticeable percentage of control mispredictions, necessitating frequent recovery actions. Therefore, it is important to exploit control independence [2], possibly by identifying threads that are non-speculative from the control point of view. When executing a control-non-speculative thread in parallel with its initiator, failure to correctly predict a branch within the initiator thread does not affect the existence of the non-speculative thread, although it can potentially affect its execution through inter-thread data dependences. Effective use of control independence information thus helps to reach distant code, despite the presence of mispredicted branches in between. Notice that if a speculative thread *T1* spawns a non-speculative thread *T2*, then *T2* is non-speculative from T1's point of view, but not from *T1*'s initiator's point of view.

Sticking to loop-based threads and non-speculative threads alone may not yield good speedup for some programs. Sometimes, it may be desirable to start a thread from a point that is control dependent on the control flow through the previous thread. This is particularly desirable when alternate control dependent paths have widely differing lengths. For example, in Figure 1(a), *Thread 2* is a speculative when spawned from basic block A, because basic block C is control dependent on block B and A. Speculative threads can also exploit more parallelism than is possible with conventional multiprocessors that lack a recovery mechanism. In fact, as we will see, for many of the non-numeric programs, speculative threads are a must for exploiting thread-level parallelism.

2.2.0.4 Out-of-Order Spawning of Threads:.

Lastly, an SpMT may or may not support out-of-order spawning of threads. If out-of-order spawning is not allowed, then all of the dynamic threads are spawned strictly in program order. If out-of-order spawning is allowed, then threads are not necessarily spawned in program order, and a single thread may spawn multiple threads. In order to avoid deadlock in such a situation, the SpMT processor may have to occasionally pre-empt some of the (sequentially younger) threads. We can also consider SpMT models with limited out-of-order spawning. In case of out-of-order depth of 1, for instance, at most one predecessor thread can be spawned after a thread has been spawned. Therefore, if the thread has to be preempted because of a predecessor thread being spawned later, the PE has to store the state of at most one other thread. Nested spawning is particularly useful to harness the parallelism present in nested loops.

Our compiler framework is very general, and supports all of the spawning strategies, including spawning from anywhere in a thread, and nesting. In our experimental section,

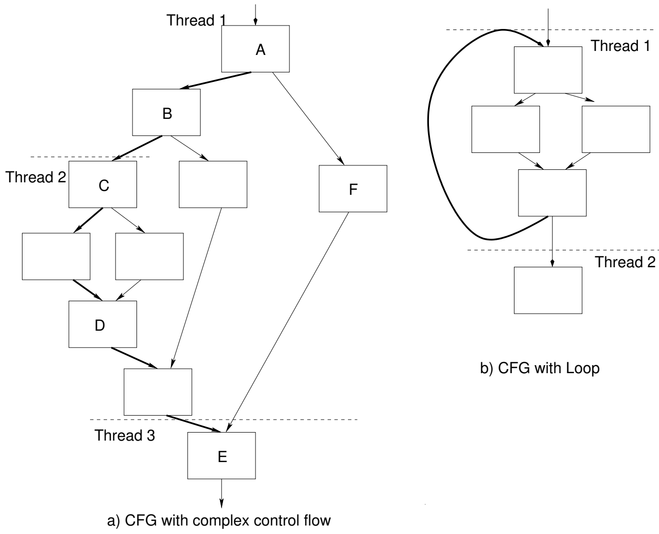

Figure 1: Different Kinds of Threads

we focus on three types of threads—*loop-centric threads, control non-speculative threads, and control speculative threads*. Figure 1 shows these types of threads.

2.3 Performance Issues in SpMT Thread Selection

Perhaps the most crucial decision in any SpMT environment is thread selection. This involves considering complex factors such as inter-thread data dependences, probability for branch misprediction within a thread, load balancing, etc.

2.3.0.5 Thread Granularity:.

Thread size is an important parameter to consider in partitioning a program into threads. Short threads may not expose adequate parallelism, and may incur high overhead depending on the thread initiation mechanisms used. Multithreading begins to make sense when threads are larger than a traditional size instruction window. On the other hand, it may not be possible to have very large size threads because of the huge buffering requirements. Moreover, if threads are very large, then recovery actions due to mispredictions will be very expensive.

2.3.0.6 Load Balancing:.

Another factor to consider in deciding thread partitioning is to reduce the variance in thread sizes. In an SpMT system, even if a particular thread is non-speculative from the control point of view, some of the data values used by that thread may be speculative, because of data dependence speculation [5], intra-thread control speculation, and possibly data value speculation [10]. Because of this speculative nature, a thread cannot be committed until all of its data operands are verified to be correct, even if its execution was completed a long time back. Of course, it is possible to initiate other threads in its hardware sequencer while the thread is awaiting retirement (as in [16]); but there is a practical limit to how many such threads can be made to wait for retirement, because of the need to store the state information of all pending threads. In short, thread size imbalance can be tolerated to some extent, but widely differing thread sizes should be avoided as much as possible.

2.3.0.7 Inter-Thread Data Dependences:.

An important factor to consider when partitioning a program into threads is inter-thread data dependences. They affect both inter-thread data communication and determine how much thread-level parallelism exists. The effect of a data dependence depends on the producer's and consumer's respective positions in their threads. It is not possible to detect all data dependences statically at compile time because of aliasing. It is also not possible to determine accurately the relative timing of the dependent instructions in different threads because of factors like conditional branches and cache misses. The compiler can use some profile information and heuristics to estimate the relative distance between the dependent instructions. The compiler can also perform intra-thread scheduling to further reduce the delay.

2.3.0.8 Thread Prioritization:.

Compilers typically do not assume a fixed number of PEs while performing thread partitioning. On a processor that has a limited number of PEs, some strategy has to be implemented to prioritize the available threads. One simple strategy is to prioritize the threads according to their sequential execution order. The motivation is that a sequentially older thread perhaps has a higher likelihood of completing earlier. This strategy is employed in the multiscalar processor [16], superthreading processor [18], and trace processor [14]. If a sequentially younger thread is both control independent and data independent of the previous threads, however, there may be merit in assigning a higher priority to it. The processor may also decide not to spawn a low-priority thread if there are not enough PEs.

Besides these major factors, thread selection also involves considering other, more subtle, factors such as probability for control misprediction within a thread.

2.4 Prior Compiler Work on SpMT

Most of the SpMT proposals advocate thread selection at compile time, because the hardware is quite limited in its program partitioning capability. There have been several proposals and implementations of compiler-based thread generation for SpMT systems [13] [18] [19]. Among these, the Agassiz compiler [18] and chip multiprocessing [13] focus on loop-level parallelism mainly. They generate threads for multiple iterations of the same loop. The Agassiz compiler also performs code scheduling within the threads, so as to facilitate pipelined execution of the threads in the superthreaded processor.

The multiscalar compiler [19] was the first major effort to partition the entire program, including the non-loop threads, for parallel execution in an SpMT processor. It uses a set of compiler heuristics to generate the threads; some of the heuristics are specific to the multiscalar architecture. For example, the multiscalar processor uses a successor thread prediction strategy, and for that each thread is restricted to have at most four successor threads. Also, the multiscalar processor does not support nested threads; so threads are spawned and initiated only in the program order. However, our compiler framework supports nested threads. For some program structures, this kind of spawning yields better performance, as will be evident from our simulation results. In the multiscalar, a successor thread is spawned only from the beginning of a thread. Our compiler supports a more relaxed spawning strategy: a thread can be spawned from anywhere within a thread. Sometimes, the spawning is delayed until a particular branch or data dependence gets resolved.

Apart from these SpMT compiler work, there has been some notable compiler work for other parallelization models. Some of the notable ones among them are the IMPACT compiler [7], the EARTH-McCAT compiler[17], and the XMT [12] compiler. The IMPACT compiler takes sequential programs, and performs a variety of optimizations, including predicated execution, superblock formation, and hyperblock formation [7]. These optimizations are geared for wide-issue uniprocessors. The focus of our compiler framework, on the other hand, is to exploit thread-level parallelism (TLP), which complements instruction-level parallelism (ILP).

The EARTH multi-threaded framework provides simple extensions to the C language, called EARTH-C [11]. This extension includes simple constructs for specifying control parallelism and data locality, which enable the programmer to specify coarse-grain parallelism. The EARTH-McCAT compiler augments this coarse-grain parallelism with fine-grain parallelism that it detects using dependence analysis. The main difference between our multi-threading framework and the EARTH framework is that the input to our compiler is a sequential program written in a standard language such as C. Furthermore, EARTH uses multithreading for hiding latencies; a long latency operation and an instruction depending on it cannot therefore coexist in the same thread. Moreover, EARTH does not support speculative execution; a thread starts execution only when its data are available, and the threads are non-preemptive. On the other hand, our SpMT framework supports preemptive threads, and a thread is speculatively executed when its data are not available.

XMT [12] is a multithreaded programming model where the programmer explicitly specifies the parallel threads. It has a simple thread execution model. The main task of the XMT compiler is to perform thread scheduling and perform the transition between the parallel and sequential environments.

One distinct feature of our compiler framework is that it starts with sequential programs written in ordinary languages, and does not require the programmer to identify or express parallelism. To the best of our knowledge, our thread generation framework is the first compiler-based thread partitioning scheme that attempts to exploit control independence and also permits nested threads.

3. COMPILER FRAMEWORK AND ALGORITHMS

In this section we present our compiler framework for partitioning sequential programs into threads. Given a program, the compiler specifies a set of thread spawning points and corresponding thread starting points. The threads share the same register name space and the same memory address space. An instruction can spawn at most one thread; a thread can collectively spawn several threads. A particular thread can also be spawned from different threads. The processor supports control speculative threads; i.e., a thread can be spawned by an instruction before knowing for sure if control flow will reach that thread. If it is found that the

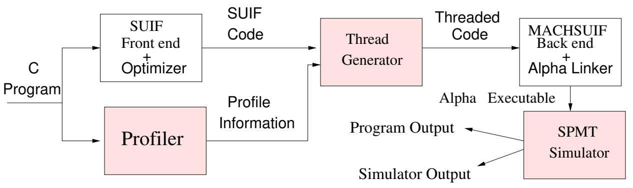

Figure 2: The Layout of the Compiler and Simulator Framework

control speculation was wrong, then the SpMT processor performs the required recovery actions.

The layout of our compiler framework, along with the SpMT simulator, is shown in Figure 2. While partitioning the program into threads, the compiler has to consider three orthogonal factors—*data dependence*, *control dependence*, and *thread size*—together, to decide a good partitioning. It employs some metrics to help in this endeavor. In the following subsections we discuss how the compiler takes care of data dependence, control dependence and the thread size. Our compiler performs the program analysis and partitioning on a high level intermediate representation. The high level representation retains all of the source level pointer and type information, and hence it is possible to take into account the dependences due to pointer aliasing. This permits more accurate data dependence information to work with. Hence the compiler is able to extract parallelism even from the pointer-intensive programs. We have used the profiling information to find out the most likely path, that the control will take and this information is used by the compiler to spawn threads speculatively.

3.1 Program Profiling

We have used a separate compiler pass to instrument the source code and gather the profiling information. In the profiling pass, we find out for every basic block, which basic block is most likely to be visited next. The compiler uses this to find out the most likely path and also to estimate the number of instructions that would be executed between two basic blocks.

3.2 Data Dependence Modeling

In our framework we have implemented two different metrics to quantify the data dependences between adjacent threads. One metric is *data dependence count* and the other is *data dependence distance*. Our thread partitioning algorithm works in multiple passes. In the first pass, the compiler builds the control flow graph (CFG) [2] and also finds out the data dependence information. It calculates the *read/write* sets [1] for every instruction. We have implemented a pointer analysis framework to obtain an improved data dependence information.

The pointer analysis helps us in getting more precise read/write sets. After calculating the read/write sets for every instruction, data flow analysis is performed. For every variable in the read set of an instruction, the set of reaching definitions [1] are determined.

3.2.1 Data Dependence Count

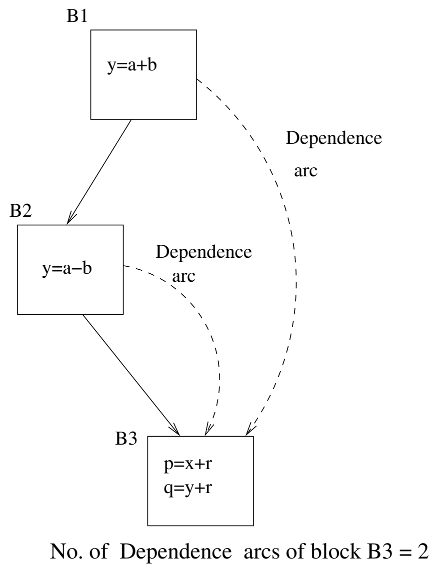

Figure 3: Data Dependence Arcs between Basic Blocks

The *data dependence count* (DDC) is the weighted count of the number of data dependence arcs coming into a basic block from other blocks. This models the extent of data dependence this block has on other blocks. If the dependence count is small then this block is more or less data independent from other blocks and we can begin a thread at the beginning of that basic block. While counting the data dependence arcs, the compiler gives more weights to the arcs coming from blocks that belong to threads that are closer to the block under consideration. The motivation is that dependences from distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for data generated there. Furthermore, the compiler gives less weightage to the data dependence arcs coming from the less likely paths. The rationale behind using the data dependence count are twofold. First of all, it is simple to compute. Also if the processing elements do out of order execution then the data dependence distant model may not be very accurate because it assumes serial execution within each thread. But in practice, due to out of order execution, instructions that are lower in the program order can be executed before the earlier instructions inside the threads. So data dependence count tries to model the extent of data dependence in the presence of out of order execution.

3.2.2 Data Dependence Distance

The *data dependence distance* between two basic blocks $B1$ and $B2$ models the maximum time that the instructions in block $B2$ will stall for instructions in $B1$ to complete, if $B1$ and $B2$ are executed in parallel. For example, consider the code segment in Figure 2. Instructions 2 and 3 of $B2$ are data dependent on instructions 1 and 5 of $B1$, respectively. If $B1$ and $B2$ are executed in parallel in two different PEs, then instruction 2 of $B2$ will not stall due to the dependence, because `x` has already been computed before instruction 2 is executed. However, instruction 3 of $B2$ has to wait for $B1$ to execute instruction 5. If we assume that every instruction has a latency of 1 clock cycle, then

[2] In a control flow graph (CFG), the basic blocks are represented by th vertices and the edges show the flow of control between the basic blocks

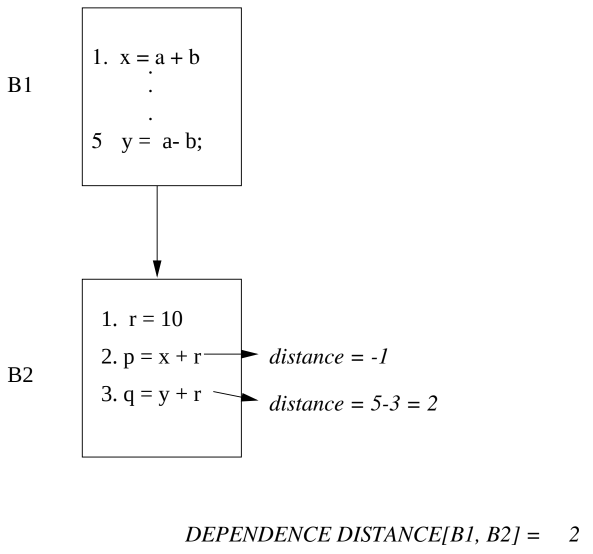

Figure 4: Data Dependence Distance between Two Basic Blocks

instruction 3 in $B2$ will stall for 2 cycles. So in this example, the maximum delay that will be encountered if $B1$ and $B2$ are executed as parallel threads is 2 cycles. Note that while computing the data dependence distance, we model that the instructions inside a single basic block are executed sequentially. Also note that the data dependence distance will increase, if the basic block $B1$ is executed as a part of a thread and there are more instructions before $B1$ and we start a new thread at the beginning of $B2$. Similarly the data dependent distance will decrease if $B1$ and $B2$ are part of the same thread and are executed sequentially. As evident from this example, it is not beneficial to execute in parallel two basic blocks with large data dependence distances. In order to decide whether to start a new thread at a control independent point, the compiler calculates the data dependence distance that will result if a new thread is started at that point. If it results in a large data dependence distance, then the compiler starts a new thread at that point.

3.3 Program Partitioning

This subsection describes the partitioning algorithm. The overview of the partitioning algorithm is given in Figure 5. The compiler partitions the CFG into multiple threads, and also annotates the instruction from which a particular thread can be spawned. In *partition_a_procedure()*, the loops are examined and partitioned first. In our compiler framework, the loops are treated as a special case of control dependence. For loops the compiler checks the dependence between two successive iterations of the loops, and if it is found that spawning another thread for the next iteration is profitable, then a thread is spawned. It may also happen that, instead of spawning from the beginning of the loop for the next iteration, the compiler spawn the next iteration from somewhere inside the loop. The large body of the loops may be further partitioned into multiple threads as described below. While partitioning the loops, the compiler uses profile information on the number of iterations and the number of dynamic instructions in the loop. Typically the compiler does not want to execute small loop body in parallel. However, if the number of iterations is large then the compiler would spawn the iterations as separate threads. Otherwise the thread will become very large. For small loops, the parallelism can be further increased by loop unrolling. For partitioning the 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 executed in parallel.

After partitioning the loops, further partitioning is done by traversing the CFG from root. At every iteration of the *do loop* in the *partition_a_procedure()* function, the compiler looks ahead till the control independent basic block of the current basic block under consideration and partitions the CFG between these to basic blocks into threads by calling the *partition_thread()* function.

The pseudocode for the implementation of *partition_thread()* function is also shown in Figure 5. *partition_thread()* takes two basic blocks and the current thread as inputs and if possible, partitions the program segment between this two basic blocks into multiple threads by calling itself iteratively. It first finds out the most likely path between the start and the end blocks of the segment by using the profile data. In *find_min_delay()* function the minimum delay is computed by using one of the data dependence models described in section 3.2. It considers only the most likely path between the two basic blocks to compute the delay. The *find_min_delay()* function looks ahead and builds a possible future thread starting at *pdom_block* using profile information and a threshold for thread size. After that it calculates the likely delay that this thread will have to suffer when it is spawned from an instruction contained in the current thread. The current thread is considered to be consisting of basic blocks from previous control independent regions and the basic blocks from the most likely path in the current region. This function also identifies the instruction in the current thread from where this future thread should be spawned in order to optimize the delay. Estimating the delay is one of the most important tasks in thread partitioning. After calculating the possible delay, the *partition_thread()* procedure goes on creating the threads. To maintain load balancing between the threads, it uses a lower limit and an upper limit for the number of instructions that can be executed as one thread. The compiler partitions the program so as to optimize the execution in the most likely path. How *partition_a_procedure()* handles load balancing and dependence delay together is explained using Figure 1.

Several cases that may arise during program partitioning are shown in Figure 1(a). The most likely path from A to E is shown by thick arrows and this likely path is quite long. So the compiler recursively looks inside the path to further partition it into smaller threads. However, if it is found that spawning a thread at E from an instruction in *Thread 1* results in a likely delay less than $DELAY\_THRESHOLD$, then the thread starting at E is spawned from *Thread 1*. In Figure 1 (a), the path between A and E is further partitioned into a thread (i.e. *Thread 2*), and this is spawned from *Thread 1*. *Thread 3* can be spawned from inside *Thread 2* or *Thread 1*, depending on the possible delay. The latter case involves out-of-order spawning. This is done in order to exploit the distant parallelism available in the program. In Figure 1(a), the region between C and D is small. If all

```
partition_a_procedure(procedure p) {
    foreach loop L in p
      partition_loop(L);
    endfor;
    start_block = p.entry_block;
    pdom_block = postdom(start_block);
    curr_thread = create_new_thread(start_block, null);
    do {
      curr_thread = partition_thread(start_block,
                              pdom_block, curr_thread);
      start_block = pdom_block;
      pdom_block = postdom(start_block);
    } while (pdom_block != null);
}

partition_thread(start_block, end_block, curr_thread) {
  pdom_block = postdom(start_block);
  path = find_most_likely_path(start_block, pdom_block);
  min_delay = find_min_delay(start_block, pdom_block,
                          path,  curr_thread, &spawn_instr);
  thread_size = path.size + curr_thread.size;
  if ( is_medium(thread_size) && ( min_delay< DELAY_THRESH))
       curr_thread.add_blocks(path);
       curr_thread = create_new_thread(pdom_block, spawn_instr);
       curr_thread = thread_partition(pdom_block, end_block,
                                        curr_thread);
        }
else if (is_big(thread_size)) {
     curr_thread.add_block(path.first_block);
     curr_thread = thread_partition(path.first_block,
                            pdom_block, curr_thread);
     if ( min_delay < DELAY_THRESH)
          curr_thread = create_new_thread(pdom_block,
                                          spawn_instr)'
     curr_thread = thread_partition(pdom_block,
                            end_block, curr_thread);
   }
else {
   curr_thread.add_blocks(path);
   curr_thread.add_block(pdom_block);
   curr_thread = thread_partition(pdom_block,
                            end_block, curr_thread);
   }
thread_partition_for_other_paths(start_block, end_block);
return curr_thread;
}
```

Figure 5: The Program Partitioning Algorithm

of the instructions belonging to the likely path between C and D are included in *Thread 2*, the size of the thread is not going to violate the upper limit. So the compiler does not spawn a new thread at D. Rather, it includes all blocks between C and D in *Thread 2* and looks beyond D to find the next potential thread starting point.

The function calls are handled automatically in the *partition_thread()* procedure. The compiler terminates the basic block after a function call. So the instructions following a function call appear in the post dominator block of the basic block containing the function call. When the compiler encounters a function call, the compiler takes into account the number of dynamic instructions to complete this function call. The compiler performs some simple inter-procedural analysis like reads and writes into the global variables and the reference parameters, to determine the possible delay. If the called function is a small one, then it is completely included in the current thread. However, for a call to a bigger function, a new thread may start executing after the function call, depending upon the possible delay and the thread size. In that case, out-of-order spawning may take place, if that function is partitioned further into threads.

The compiler also checks the paths that are not the likely paths and partitions them as well. If at run-time, control goes into those unlikely paths, then the threads spawned speculatively are aborted. But the threads that are not control dependent on the aborted threads need not be aborted. For example, consider Figure 1 (a). If from A, instead of following the most likely path, the control goes to basic block F, when both threads 2 and and 3 have been spawned, thread 2, would be aborted, but not thread 3, as E is control independent of A.

3.4 Implementation Overview

Our compiler framework is implemented on the SUIF-MachSUIF platform [6]. The layout of the complete framework is shown in Figure 2. All of the compiler analysis and thread partitioning are done at the high-level intermediate representation (IR) of SUIF. We have chosen the SUIF platform to implement our compiler system because it provides a modular and flexible infrastructure to develop compiler optimizations. SUIF first translates high-level source code into an IR, and then performs code optimization through several independent passes on that IR. We find it easier to work with the SUIF IR, and to integrate our own compiler passes in that framework. While transforming high-level programs into IR, SUIF retains all of the relevant information from the high level source program. This is particularly helpful for carrying out optimization such as pointer analysis. Therefore, the compiler can perform more accurate program analysis. Moreover, the instructions in the SUIF IR are very close to the assembly level instructions; thus, the estimation of thread sizes done at IR level remains valid in the final assembly level as well. In SUIF, it is possible to annotate the instructions with necessary information like data dependence, and use them in separate passes afterwards. Also, the SUIF package contains many optimization modules, which improve the quality of the code produced.

We used the MachSUIF [15] framework to generate Alpha assembly code from the SUIF IR. We have implemented the profiling phase also in the SUIF framework.

4. EXPERIMENTAL EVALUATION

To study the effectiveness of our thread partitioning schemes, we conducted a simulation-based evaluation. This section details the simulation framework and the simulation results obtained.

4.1 Experimental Setup

4.1.1 Experimental Methodology

The central goal of these experiments is to understand the potential of different thread partitioning algorithms. Our objective is not to evaluate the performance of a specific (multithreaded) microarchitecture. While using a detailed microarchitecture simulator, if the performance is poor, we gain little insight on why it does not work, or whether it is the thread partitioning scheme or machine model (or both) that should be improved. As a consequence, poor results may not reflect on any inherent limitations of the thread partitioning algorithm, but rather the way it was applied in a microarchitecture. To search through a large space of thread partitioning schemes effectively, we use a trace-driven simulator. If a partitioning scheme does not work well with this simulation framework, it will not work well on any real processor of a similar design.

This experimental analysis serves an important function in showing the limits of certain thread partitioning algorithms, such as parallelizing only loops, and recognizing issues that are worthy of further attention. Our SpMT simulator models a multi-threaded processor on top of a trace-driven simulator. The modeled SpMT processor consists of multiple processing elements (PEs). Each PE has its own program counter, fetch unit, decode unit, and execution unit, to fetch and execute instructions from a thread. The PEs are connected together by an interconnection network. The number of PEs, issue size per PE, etc., are parameterized. The simulator uses the Alpha ISA. For the sake of simplicity, we assume that each PE takes one cycle to execute each instruction. We model a memory hierarchy with a shared L1 d-cache with 1 cycle latency and a memory access latency of 10 cycles. When encountering a conditional branch instruction in a thread, its PE consults a branch predictor for making a prediction. We also model a hybrid data value predictor [20] for predicting the results of instructions whose operands are unavailable at the time of fetching.

The code executed in the supervisor mode are unavailable to the simulator, and are therefore not taken into account in the parallelism studies. The library code is not parallelized, as we use the standard libraries in our experiments. The library code therefore executes in serial mode, providing a conservative treatment to our parallelism values.

4.1.2 Hardware Parameters Used

For our simulation we have used a PE issue width of 4 instructions per cycle and the PEs use out-of-order issue. Each PE has an instruction window of 128 instructions. The L1 cache size is 256 Kbytes. There is a 2-cycle overhead in assigning a thread to a PE and thread pre-emption also incurs a 2 cycle penalty. Furthermore, it assumes a 2-cycle latency for forwarding register values across multiple PEs.

4.1.3 Benchmarks

Table 1 lists the benchmark programs used for the evaluation of the compiler framework. We have used five programs from SPEC2000, one from SPECINT95, and six from the Olden benchmark suite. All of these programs are written in C. Our multi-threading compiler framework partitions into threads all of the source code, except the library code and the system code. Each benchmark is executed for 300 million instructions, except for `perimeter`, which completed execution after 89 million instructions. For SPEC benchmarks we have used the *train* data sets as inputs. Most benchmark programs spend some time in the beginning for initializing data structures and reading inputs, and these parts of the programs do not reflect the actual program characteristics. So we have used a "fast forward" mode to skip these initialization phases, after which the statistics are collected. The number of instructions that have been fast forwarded are shown in Table 1.

4.1.4 Default Partitioning Setup

As there are many different parameters, it is difficult to perform a completely orthogonal set of experiments. Instead, we define a default setup, and vary one parameter at a time. Thus, when the nature of threads is varied, the rest of the parameters are kept at their default values. For the default configuration, we allow all kinds of threads (i.e., *speculative* threads, *control independent* threads, and *loop based* threads), data dependence distance based modeling of inter-thread data dependences, and data value prediction.

4.2 Effectiveness of the Partitioning Algorithm

To evaluate the effectiveness of our partitioning algorithm, we measure the speedup obtained by increasing the number of PEs from 1 to 6 with our default configuration. Figure 6 shows the speedup obtained over a single PE. In the figure, each bar along the X-axis represents a benchmark program and the Y-axis represents the speedup over single PE. Table 2 presents some thread-related statistics for the default configuration.

The speedup with 6 PEs ranges from 1.62 for `health` to 4.68 for `mst`. Most of the benchmarks show good speedup and scalability as we increase the number of PEs. *crafty* spends most of the time outside loops[3] and the fact that it shows good speedup and scalability suggests that the compiler has been able to extract parallelism from *non-loop* parts of the code effectively. This is true for the other benchmarks like `vpr`, `perimeter`, `power`, `tsp`, and `treeadd` as well. `perimeter` and `treeadd` do not have loops; they have recursive function calls instead. All these benchmarks execute a large percentage of *speculative* and *non-speculative* threads.

Benchmarks `ijpeg`, `mcf`, `twolf`, and `health` show modest speedups. The scalability is also quite low. In `ijpeg`, `mcf`, `twolf`, and `health`, most of the time is spent in loops, and these loops have a large number of loop-carried dependences. So these programs only show moderate speedups with multithreading. Moreover, we see from Table 2 that the average number of dynamic instructions per thread for `health` is only 8.89, which is quite low. Therefore, in `health`, the PEs are not able to exploit thread-level parallelism well, which accounts for its modest speedups and poor scalability. On average, we get a speedup of 2.89 with 6 PEs.

From Table 2 we see that except for `mst` and `health`, the average thread sizes are also reasonable. In `health` there is a small loop body that is getting executed in parallel most

[3] In this context, by *loops*, we do not mean those loops where loop bodies contain function calls such that successive iterations of the loops are thousands to millions of instructions apart, e.g., the processing loop in the *main()* function

Benchmark Suite	Program Name	Description	Lines of Source Code	No. of Instrs Fast Forwarded
SPEC 95	ijpeg	Compresses and Decompresses ppm file	28566	250000000
SPEC2000	crafty	Chess Program	20294	100000000
	equake	Finite element simulation: earthquake modeling	1513	75000000
	mcf	Minimum cost network flow solver	1909	100000000
	twolf	Place and route simulator	19762	500000000
	vpr	Circuit placement and routing	16973	150000000
Olden	health	Columbian Health Care Simulator	505	0
	mst	Minimum Spanning Tree	417	27000000
	perimeter	Quad Tree	290	0
	power	Power Pricing Problem	616	0
	treeadd	tree traversal Problem	121	0
	tsp	Traveling Salesman Problem	521	0

Table 1: Benchmark Programs

Program Name	Avg. Thread Size (Dyn. Instrs)	Thread Type		
		Speculative	Non-speculative	Loop-centric
ijpeg	75.67	21.01%	0.65%	78.32%
crafty	81.55	56.27%	11.05%	32.68%
equake	27.99	0.50%	0.80%	98.70%
mcf	33.20	0.15%	0.07%	99.78%
twolf	33.46	4.17%	3.21%	92.61%
vpr	83.77	28.95%	17.05%	53.99%
health	8.89	0.50%	0.00%	99.50%
mst	574.07	0.00%	0.00%	100.00%
perimeter	105.88	87.72%	12.28%	0.00%
power	42.62	6.47%	71.69%	21.84%
treeadd	106.48	99.99%	0.01%	0.00%
tsp	102.84	11.08%	0.13%	88.78%

Table 2: Thread Statistics

of the time resulting in small threads. On the other hand, in `mst`, the *loop-centric* thread that is getting executed most of the time contains library routine calls that our compiler did not partition, resulting in very large thread size.

4.3 Experimentation with Thread Type

Our next set of experiments focus on varying the nature of threads. In particular, we simulate three different combination of threads: (i) loop-based threads, non-speculative threads, and speculative threads – i.e our default configuration; (ii) loop-based threads and non-speculative threads; and (iii) loop-only threads. Figure 7 compares (i) and (ii) and (iii). In this figure, the X-axis denotes the benchmarks, and the Y-axis denotes the speedup with 6 PEs. For each benchmark, three bars are shown, corresponding to the three different combinations of threads. We have tried to manually validate that *loop-centric* thread partitions are indeed the good ones. It is not feasible to do that manually for the other kinds of threads.

On analyzing the results of Figure 7, we can see that loops-only threads are quite insufficient to harness the parallelism present in `crafty`, `vpr`, `perimeter`, `power`, and `tsp`. As mentioned earlier, `perimeter` and `treeadd` do not contain any loops. Moreover, from Table 2, we find that they primarily consist of *speculative* threads. So it is not surprising to see that their performance does not improve even after including *control independent* threads with *loop-centric* threads. Both these programs have recursive function calls and the

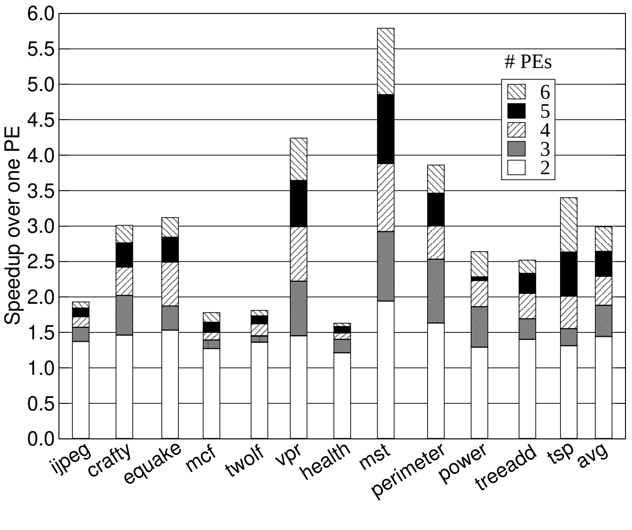

Figure 6: Speedup with Varying Number of PEs

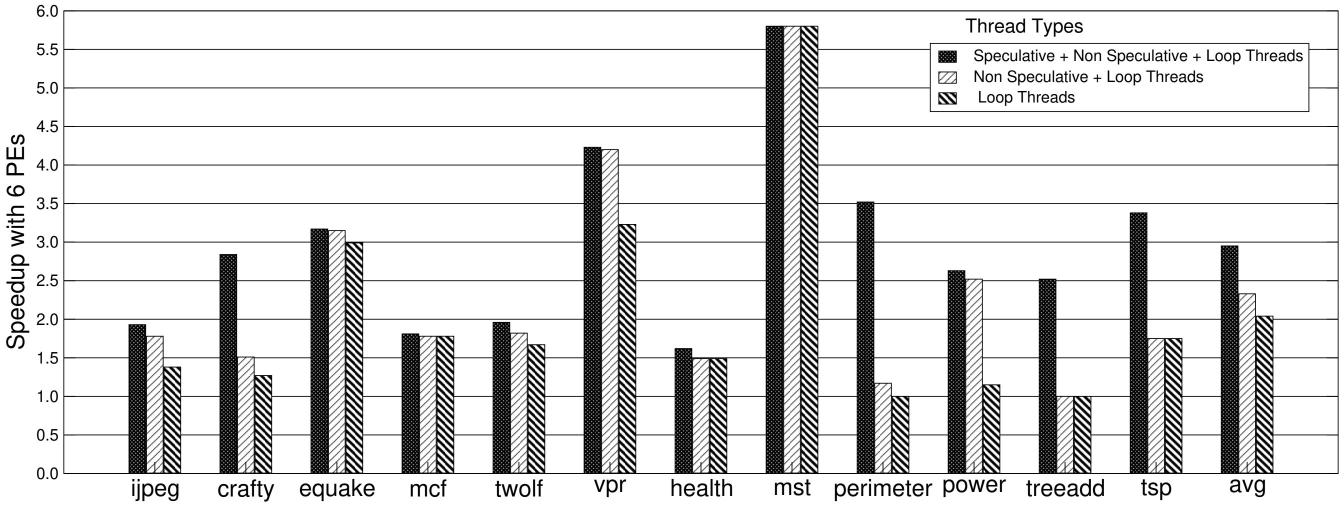

Figure 7: Speedups with Different Types of Threads

functions are called conditionally. These function calls can be executed in parallel and by executing them speculatively it is possible to get large parallelism. In `crafty`, only a little time is spent in the loops, and also the loops are not quite parallelizable. So we get small speedup with *loop-centric* threads only. From Table 2 we see that more than 50% of the threads are *speculative* threads and so *control-independent* threads along with *loop-centric* threads could not exploit all the available parallelism in the program. In `tsp`, although only 11% 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. In `power`, 72% of the threads are *non-speculative* and only 6% are *speculative*. So by executing *non-speculative* threads along with *loop-centric* threads, it is possible to achieve complete speedup. Benchmarks `equake`, `mcf`, `health`, and `mst` spend most of the time in parallelizable loops. So these programs are able to harness almost all of the available parallelism by executing the *loop-centric* threads only. Although `ijpeg` and `vpr` contain a significant percentage of *speculative* threads, the results show that it is possible to exploit all of the available parallelism without using them. This is because the load balancing remains unaffected even after ignoring the *speculative* threads and the scheduling also do not get affected adversely. Moreover, the ILP gets boosted in the bigger threads resulting a good speedup.

4.4 Experimentation with Data Dependence Modeling

Our next set of experiments focus on the type of data dependence modeling used by the compiler while deciding thread partitioning. In particular, we look at two models: one based on data dependence count and the other based on data dependence distance. Figure 8 presents these results; these results are a mixed bag. For `ijpeg`, `vpr`, `mst`, and `perimeter`, data dependence distance-based modeling gives better parallelism, and for `crafty`, and `mcf`, `twolf`, and `treeadd` it is just the opposite. For other benchmarks, the speedups are almost the same. Except for `perimeter` and `vpr` in all other cases the differences in speedups are not appreciable. On looking into the partitioning done for `perimeter`, we found that the count based modeling was conservative and failed to identify a partitioning opportunity. It honored a data dependence and restrained from partitioning, whereas the distance based modeling ignored that dependence because it estimated that the subsequent threads did not have to wait for it. At runtime this data dependence did get resolved early, and so the performance of the latter partitioning becomes much better than the former one. From the results, we see that both the models are quite effective in representing the data dependence in the programs.

4.5 Effect of Out-of-Order Spawning

Our last set of experiment focus on the effect of out-of-order thread spawning. Our compiler framework can theoretically support 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. Also, in order to support out-of-order thread spawning, the SpMT processor may have to frequently preempt some of the (sequentially younger) 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 this set of experiments, we compare the speedups obtained with 4 different depths of out-of-order spawning: (i) sequential spawning only, (ii) out-of-order spawning depth of 2, (iii) out-of-order spawning depth of 4, and (iv) out-of-order spawning depth of infinity. The default configuration assumes that the PEs can buffer an infinite number of successor threads.

The results are shown in Figure 9. Benchmarks `ijpeg`, `mcf`, `twolf`, `health`, and `mst` show no change in speedup with nesting. This implies that even in the default configuration, the threads are spawned and executed in sequential order. Benchmarks `crafty`, `vpr`, and `tsp` show a small improvement with out-of-order spawning. In the case of `equake`,

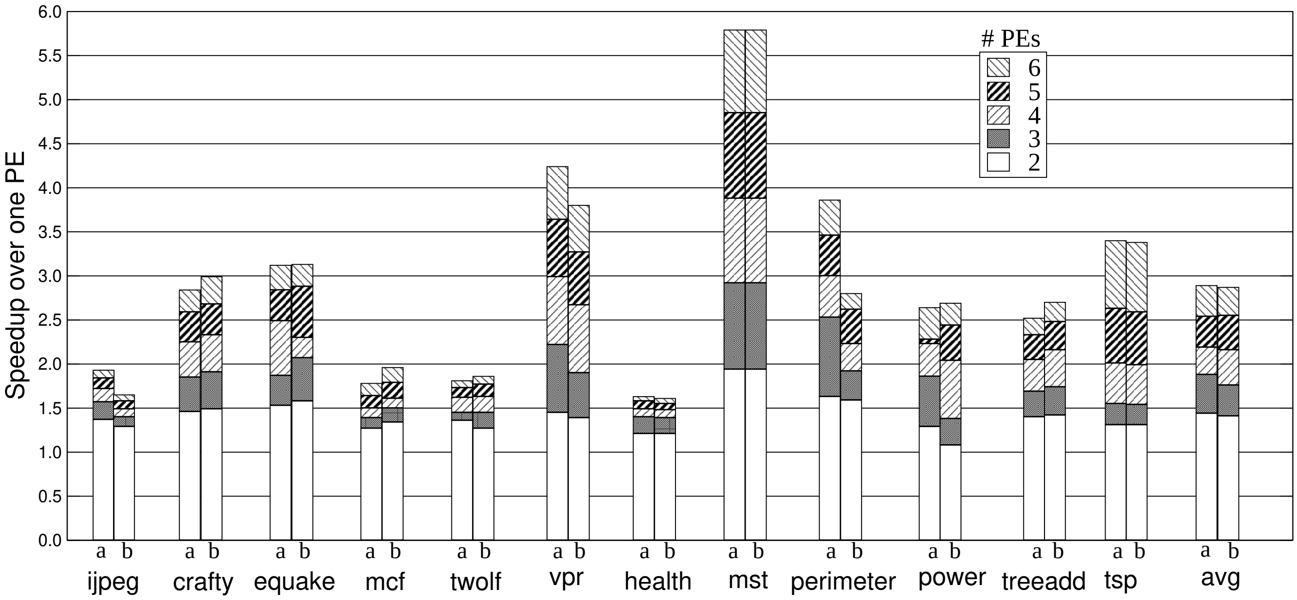

Figure 8: Speedups with Different Dependence Modeling a: Data Dependence Distance; b: Data Dependence Count

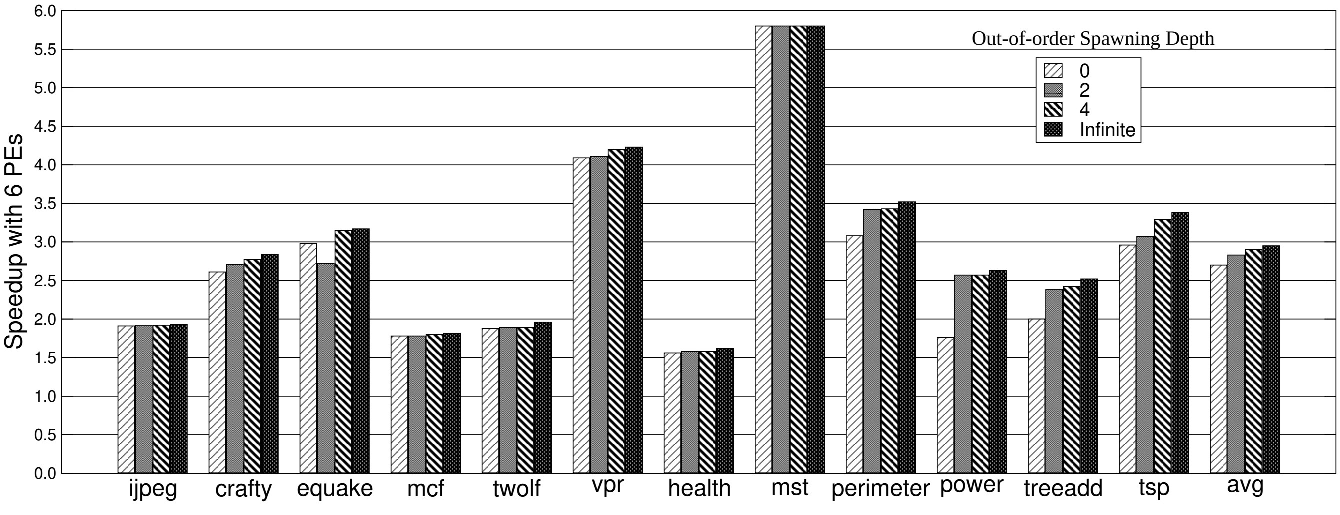

Figure 9: Speedups with different Out-of-Order Spawning Depths

there is a drop of performance for a depth of 2, and then it again goes up. This is because in `equake`, with out-of-order spawning depth of 2, the pre-emption cost overrides the advantage of having out-of-order spawning of depth 2, thereby lowering the speedup. In `power`, `perimter`, and `treead` there is significant increase in speedup even at depth 2. The increase in speedup is maximum for `power`. In `power`, the program spends about 17% time in a big loop that cannot be parallelized because of the size and data dependence. However, the loop body contains calls to functions that can be executed in parallel. The first function called is again partitioned into two threads. With sequential spawning, the second function starts execution only after the second thread of the first function starts executing. However, by allowing an out-of-order spawning depth of 1, the second function can be executed in parallel with the first function, resulting to a significant improve in performance.

5. CONCLUSIONS

Speculative multithreading (SpMT) is emerging as an important parallelization tool for non-numeric programs. Examples are the multiscalar processor [4] [16], the SPSM processor [3], and the decoupled control flow processor [8]. All of these use multiple hardware sequencers to fetch and execute multiple threads in parallel. Given the increasing interest in mainstream microprocessor design, we expect that future processors will attempt to execute multiple threads in one way or another.

Judicious partitioning of a program into threads involves a lot of analysis, which makes it difficult to be done in hardware. Previous compiler efforts have focused on identifying loop-based threads and speculative threads. A limitation of this approach is that branch mispredictions may cause all of the subsequent threads to be discarded, without retaining any control-independent threads that may be present in the processor. The use of non-speculative threads has the potential to extract additional amounts of parallelism, especially for non-numeric programs.

This paper presented a general compiler framework for partitioning a sequential program into multiple threads for execution in a SpMT processor. Our compiler framework is geared for identifying loop-based threads, speculative threads, and non-speculative threads. In addition, it also supports nested 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.

We have implemented this compiler framework on the SUIF-MachSUIF platform. Our framework is is able to partition into threads large programs, such as the SPEC benchmark programs. A simulation-based evaluation of the generated threads indicate that an average speed up of up to 3 can be obtained with 6 processing elements for SPEC INT programs and Olden programs by using speculative multithreading. This is very promising, given that non-numeric programs are inherently difficult to parallelize. Our detailed experimental analysis has increased our understanding of the different factors that affect performance. These analyses show that the combination of loops, speculative, and non-speculative threads has the potential to extract thread-level parallelism in non-numeric programs.

6. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools. *Addison-Wesley,* Reading, MA, 1986.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[3] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading, *Proc. International Conference on Parallel Architecture and Compilation Techniques (PACT '95)*, 1995.

[4] M. Franklin, The Multiscalar Architecture, *Ph.D. Thesis, Technical Report 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[5] M. Franklin and G. S. Sohi, ARB: A Hardware Mechanism for Dynamic Reordering of Memory References, *IEEE Transactions on Computers*, Vol. 45, No. 5, pp. 552-571, May 1996.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[7] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler Technology for Future Microprocessors. *Proc. IEEE*, 83(12):1625–1640, December 1995.

[8] S. Jayashree and S. Vajapeyam, Exploiting Parallelism across Basic Blocks via Decoupled Control Flow, *Technical Report TR No. IISc-CSA-95-01*, Department of Computer Science and Automation, Indian Institute of Science, March 21, 1995.

[9] R. Joy and K. Kennedy. *President's Information Technology Advisory Committee (PITAC) - Interim Report to the President.* National Coordination Office for Computing, Information and Communication, 4201 Wilson Blvd, Suite 690, Arlington, VA 22230, August 10, 1998.

[10] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. *Proc. 19th Annual International Symposium on Computer Architecture*, 46–57, 1992.

[11] O. C. Maquelin, H. H. J. Hum, and G. R. Gao. Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors. *Proc. First International EURO-PAR Conference*, 1995.

[12] D. Naishlos, J. Nujman, C.-W. Tseng and U. Vishkin, Evaluating Multi-threading in Prototype XMT Environment, *Proc. 4th Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC-2000)*

[13] K. Olukotun, et al. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, September 1999.

[14] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, Trace Processors, *Proc. 30th International Symposium on Microarchitecture*, pp. 138-148, 1997.

[15] M. D. Smith and G. Holloway. An Introduction to

Machine SUIF and Its Portable Libraries for Analysis and Optimization.

[16] G. S. Sohi and S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. *Proc. 22nd International Symposium on Computer Architecture (ISCA)*, 414–425, 1995.

[17] X. Tang. Compiling For Multithreaded Architectures *Ph.D. Thesis*, Department of Electrical Engineering, University of Delaware, 1999.

[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. on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[19] T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. *Proc. 31st International Symposium on Microarchitecture (MICRO-31)*, 1998.

[20] K. Wang and M. Franklin, Highly Accurate Data Value Prediction using Hybrid Predictors, *Proc. International Symposium on Microarchitecture (MICRO-30)*, pp. 281-290, 1997.