Abstract

It is well known that a large fraction of applications cannot be parallelized at compile time because of unpredictable data dependences due to indirect memory accesses and/or memory accesses guarded by data-dependent conditional statements. A significant body of prior work attempts to parallelize such applications using runtime data-dependence analysis and scheduling. Performance is highly dependent on the ratio of the dependence analysis overheads with respect to the actual amount of parallelism available in the code. We have found that the overheads are often high and the available parallelism is often low when evaluating applications on a modern multicore processor.

We propose a novel software-based approach called dependence-aware scheduling to parallelize loops with unknown data dependences. Unlike prior work, our main goal is to reduce the negative impact of dependence computation, such that when there is not an opportunity of getting speedup, the code can still run without much slowdown. If there is an opportunity, dependence-aware scheduling is able to yield very impressive speedup.

Our results indicate that dependence-aware scheduling can greatly improve performance, with up to 4x speedups, for a number of computation intensive applications. Furthermore, the results also show negligible slowdowns in a stress test, where parallelism is continuously detected but not exploited.

Keywords: Partial Parallelism, Runtime Dependence Analysis, Inspector/Executor, Multicore, Thread Scheduling.

1. Introduction

Current parallelizing compilers aim to extract parallelism from programs with regular, statically analyzable memory access patterns. However, there are a significant number of applications that have memory access patterns that are not readily analyzable at compile time. For instance, pointer dereferences or indirect memory references are often difficult to capture through static analysis; array subscripts can involve computations that cannot be resolved statically; and conditional branches can selectively expose or hide memory accesses, leading to dynamic memory access sequences.

Thread Level Speculation (TLS) [2][3][4][5][23] aims to parallelize codes with potential memory access conflicts using dedicated hardware mechanisms to detect data-dependence conflicts and to roll back state when violations occur. TLS does not require intensive compiler analysis, either statically or at runtime, and can be applied to arbitrary code. However, it requires specialized hardware support that is not yet available on commodity multicores.

Software-based approaches for Thread Level Speculation have been proposed [16][17][18][19][20][21][22], where data-dependence conflict detection mechanisms are purely implemented in software. When conflicts are detected, the state of one or more threads is rolled back, again using software. Compiler techniques can be used to optimize and reduce the conflict checks and rollbacks. While software TLS can be successful, its overheads make it generally ill suited for applications that have frequent, unpredictable dependences among consecutive iterations.

A significant body of work [6][7][8][9][10][11][12][13][14][15] has proposed runtime techniques based on detecting data-dependences and exploiting available parallelism by scheduling work accordingly. Like TLS, the performance of such techniques is highly dependent on the ratio of the cost associated with data dependence analysis and the benefit achieved by exploiting parallelism. Unfortunately, many applications do not currently benefit from these techniques on current multicores, because dependence computation overheads are often too high compared to the sequential execution time of the original code, and/or the applications exhibit only limited amounts of parallelism. The causes for high overheads can be classified as follows. First, dependence analysis often utilizes large data structures to precisely track all (possible cross-iteration) memory references and/or memory location touched by a loop, resulting in memory footprints that can be much larger than that of the original code. Since today’s multi-threaded multicores share large fractions of their memory hierarchy, they are particularly sensitive to larger memory footprints. Second, the computing of dependences often involves synchronization and/or expensive sorting algorithms. Third, sometimes the actual computations only start once data dependence computations are completed.

As a result of these high overheads, prior work is only applicable to loops with data dependences that are complicated enough so that the compiler cannot analyze them at compile time, but simple enough so that the dependence overheads are not too high. Also, the loop must run long enough, and be parallel enough, so as to amortize the initialization overhead. As a result, these drawbacks can greatly offset the benefits or even cause significant slowdown when dealing with code without prior knowledge on their dependences.

In this paper, we present a novel software-based approach called dependence-aware scheduling to parallelize code with unknown dependences. Our main goal is to significantly reduce the negative impact of dependence computation, so that (1) when there is no parallelism, the code can still run with nearly no slowdown; and (2) when there is parallelism, dependence-aware scheduling can yield significant speedups.

In our approach, a main thread runs the original code sequentially, without regards to the results of the dependence computation. Meanwhile, a number of worker threads are used to calculate dependences. A slice function is derived from the...
original code such that only a subset of the computation is performed during dependence calculation.

To minimize the impact to the main thread, lightweight data structures are used for dependence computation. The dependence computation algorithm computes a safe approximation of the original data-dependence, is completely parallel, and requires no synchronizations. As a result, resources on many cores/threads can be utilized, resulting in less interference among the threads. By fully parallelizing dependence computations, we do sacrifice some accuracy of the dependence information. However the loss of accuracy does not compromise the correctness of the execution. In other words, the computed dependences may overly constrain the scheduler, causing some loss of parallelization opportunities, but will not lead to erroneous results.

Once the dependence computation has completed, worker threads are assigned to execute some of the iterations in parallel with the main thread, as long as the iterations can be executed ahead of time. The main thread will later skip those iterations that have already been executed. In addition to the worker threads and main thread, a scheduler thread is used to schedule iterations for ahead-of-time execution. The scheduler thread is also in charge of the coordination of other threads to generate a correct execution. The separate execution of the main thread and various approaches to reduce the negative impact on it help the main thread to make steady progress even if dependence computation turns out to be futile.

Our results based on a number of computation intensive applications indicate that dependence-aware scheduling can greatly improve performance (up to 4 x speedups) when conflicts rarely happen. Our results show negligible slowdowns in a stress test, where dependence is continuously analyzed on assist threads (to exert maximal pressure on the main thread) but where the exploitation of the discovered parallelism is explicitly disabled.

The rest of the paper is organized as follows: Section 2 gives the motivation; Section 3 provides an overview of the approach; Section 4 describes how to derive the dependence slice function; Section 5 details the computation of approximate dependences; Section 6 addresses the scheduler and communication; Section 7 provides details on implementation; results are shown in Section 8. Section 9 discusses the related work and Section 10 concludes the paper.

2. Motivation

2.1 Exploit Parallelism

Many factors prevent code from being parallelized. Due to various limitations of static compiler analysis and the lack of user specifications, parallelization opportunities could be lost for code that is completely or partially parallelizable. These factors can be summarized as follows.

It’s well known that compiler optimizers make conservative assumptions to ascertain that code is generated correctly. Even if a dependence occurs only once at runtime, the compiler has to assume its existence, which prohibits many optimizations. For example, indirect accesses through pointers or array subscripts with indirection or complicated computation are often major obstacles for the compiler to determine cross-iteration dependences.

On the other hand, while the user may have a great deal of knowledge about the likely behavior of the application, it is rare that this knowledge is made available to the compiler due to the lack of expressiveness in the programming language. As a result, the runtime behavior of a program is often much more constrained than is strictly necessary.

Once the dependence computation has completed, worker threads are assigned to execute some of the iterations in parallel with the main thread, as long as the iterations can be executed ahead of time. The main thread will later skip those iterations that have already been executed. In addition to the worker threads and main thread, a scheduler thread is used to schedule iterations for ahead-of-time execution. The scheduler thread is also in charge of the coordination of other threads to generate a correct execution. The separate execution of the main thread and various approaches to reduce the negative impact on it help the main thread to make steady progress even if dependence computation turns out to be futile.

Our results based on a number of computation intensive applications indicate that dependence-aware scheduling can greatly improve performance (up to 4 x speedups) when conflicts rarely happen. Our results show negligible slowdowns in a stress test, where dependence is continuously analyzed on assist threads (to exert maximal pressure on the main thread) but where the exploitation of the discovered parallelism is explicitly disabled.

The rest of the paper is organized as follows: Section 2 gives the motivation; Section 3 provides an overview of the approach; Section 4 describes how to derive the dependence slice function; Section 5 details the computation of approximate dependences; Section 6 addresses the scheduler and communication; Section 7 provides details on implementation; results are shown in Section 8. Section 9 discusses the related work and Section 10 concludes the paper.

2.2 Prior Work

As mentioned earlier, a large body of prior work has attempted to perform runtime dependence analysis and detection. Good performance was reported for some specific benchmark suites such as the PERFECT benchmark suite. However, by and large, traditional runtime dependence analysis and detection often runs slower than the original code when applied to modern benchmarks on today’s multi-threaded multicores.
3. Overview

In this section, we provide an overview of the dependence-aware scheduling scheme that can exploit runtime parallelism and reduce worst-case slowdown. As mentioned earlier, this is a purely software-based scheme, meaning that it can be applied to any multicore machine.

Figure 2 illustrates the overall framework. Without loss of generality, we assume that the code being executed is a loop. In Figure 2, each iteration of the loop is depicted as a block labeled by its iteration ID.

There are four types of threads running in the system:
1. One Main Thread,
2. A pool of Worker Threads,
3. A pool of Dependence Computation Threads, and
4. One Scheduler Thread.

To minimize thread initialization overhead, all threads are started at the beginning of the execution and waiting for work to be assigned. The main thread executes the original loop sequentially, ensuring steady progress regardless of the outcome of the dependence computation. Before executing each iteration, it also checks whether this iteration has already been executed. If the iteration has been executed by a worker thread, the main thread simply skips it. The more iterations the main thread skips, the higher the speedup. To amortize the checking overhead, we group iterations into larger iteration blocks, such that the cost of the checking code in the main thread is negligible.

![Figure 2. Overview of the dependence-aware scheduling framework.](image)

Worker threads are used to assist the main thread by computing dependences and by executing some iterations in advance. Typically, only a subset of the worker threads are used for dependence computation. Threads in this subset are called Dependence Computation Threads (or Dep. Threads). In other words, some worker threads are first used to compute dependences, and later reassigned to execute iterations.

In Figure 2, the main thread is on the left. Bars in the top box denote dependence computation on the dep. threads. Once dependence computation finishes, some iterations e.g. Iterations 10, 13 and 20 are executed in parallel, ahead of the main thread.

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>#spec execs</th>
<th>#exec w/ conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equake</td>
<td>137</td>
<td>34</td>
</tr>
<tr>
<td>art</td>
<td>2527</td>
<td>0</td>
</tr>
<tr>
<td>wupwise</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ammp</td>
<td>250</td>
<td>203</td>
</tr>
<tr>
<td>timap</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>mcf</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>SPEC2006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>namd</td>
<td>14918</td>
<td>14918</td>
</tr>
<tr>
<td>milc</td>
<td>3969</td>
<td>0</td>
</tr>
<tr>
<td>SEQUOIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lammpsSmk</td>
<td>4000</td>
<td>479</td>
</tr>
<tr>
<td>CrystalMk</td>
<td>10000000</td>
<td>183217</td>
</tr>
<tr>
<td>IRSmk</td>
<td>57526</td>
<td>0</td>
</tr>
<tr>
<td>SphotMk</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>AMGMk</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>UMTMk</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 1. Parallelization opportunities in several benchmark suites.**

In Table 1, we show results collected for a number of benchmarks from SPEC2000, SPEC2006 and the SEQUOIA [1] benchmarks. The second column shows the dynamic number of loops with one or more data dependences that cannot be statically determined. The third column indicates the dynamic number of such loops that experienced actual data dependences at runtime. Table 1 demonstrates that for all but 4 benchmarks, chances of encountering runtime data-dependence are significant. As a result, there is no clear cut solution to decide whether or not runtime dependence analysis should be applied without strong knowledge of the runtime data-dependence patterns. It is likely that such techniques can experience significant worst-case performance, when their benefits are nullified after paying the high cost of detecting data-dependences.

The worst-case performance of software-based thread level speculation [16][17][18][19][20][21][22] can be high, because once rollbacks and re-executions happen, the code could run much slower than the original one.

Runtime data dependence analysis often requires large data structures to precisely store all possible cross-iteration memory accesses [6][9][10][11][12][13][15]. This necessitates large memory space to hold those data. Consequently, cache performance suffers when manipulating large data sets. In our experiments, we did a test that simply stores the addresses of all possible cross-iteration memory accesses in a table and reads them back using 16 threads. For most benchmarks listed in the results section, with the notable exception of IRSmk, this process took longer than the execution time of the original code. Some prior work [7][8][14] does not store all accesses, however they either involve many redundant computations [7][14], or experience severe slowdown when accesses contain hotspots (i.e. accesses to the same memory location) [8]. In general, if the dependences are intense, none of the prior studies can offer a graceful degradation. Acknowledging the overhead of dependence computation, most prior researchers [11][13][6][15] have suggested schedule reuse, i.e. storing the schedule for use during later runs of the same loop. However, it is often nontrivial to ensure that data dependences will remain the same between consecutive executions of a loop, esp. for loops with large arrays and complicated control flows. For example, the loop given in Figure 1 is highly volatile, with dependences changing slightly during each consecutive execution.
When the main thread reaches the point at which it should execute Iteration 10, it notices that Iteration 10 has already been assigned and executed by a worker thread. Therefore, the main thread skips Iteration 10. Such scheduling is based on the dependence information gathered during dependence computation, thus the correctness of ahead-of-time execution is guaranteed. Similarly, the main thread skips Iterations 13 and 20.

Dependence computation attempts to achieve the following two goals. First, it tries to finish as quickly as possible. Since the main thread starts execution right away, parallelization opportunities are lost for any iterations computed by the main threads until data dependence information is available. In the worst case, when dependence information cannot be computed before the end of the main thread, no scheduling is possible even if parallelism is detected. Second, dependence computation tries to impose as little resource pressure as possible on the concurrently executing main thread by reducing resource requirements and by being maximally parallelized.

A critical component of dependence-aware scheduling is that it engages an approximate algorithm that spreads the dependence computation to a large number of parallel threads in order to shorten the dependence computation stage. A slice function is used to extract only relevant memory accesses from the original code. This further reduces the amount of computation for dependence computation. On each thread, several approximate dependence computation algorithms have been proposed to strike a balance between accuracy and overhead. In other words, it is acceptable to follow a schedule that is more constrained than what is permitted by the actual, dynamic data-dependences.

The scheduler first waits for the completion of dependence computation, then schedules iterations according to the dependence information. It also communicates with all the other threads to avoid any deadlocks and violation of dependences. Synchronization mechanisms are deployed among threads to make sure that data is communicated properly.

### 4. Deriving the Dependence Slice Function

The Dependence Slice Function is a distillation of the loop body that only computes information that is relevant for dependence computation. Note that, unlike some of the prior work such as [9][10][11][12][13][14] that only focuses on arrays, the compiler first identifies all memory accesses that could involve dependences. This typically includes: (1) accesses through pointer operations (2) array accesses that may have loop carried dependences (3) global variables. Second, the statement is replaced with the address expression. The rest of the statement is discarded. Next, a backward slice [24] is started from each address expression. All statements on the union of all backward slices are marked. Finally, all the other statements can be safely removed, since they are not relevant to inter-iteration address accesses.

Figure 3 gives an example, which is taken from SPEC2000 equake, in function smvp. The original loop is shown in Figure 3.a. Arrays Aindex, A, v are read-only arrays, therefore they should not cause dependences. Array w could cause a dependence. It is accessed in line 13 and line 16. Since one of the indices col is read from an array, it is not possible to establish at compile-time whether this dependence will actually occur at runtime.

To construct dependences, we are only interested in the indices of w. Therefore, two backward slices are built from line 13 and line 16, with regard to col and i. All right-hand side expressions are safely removed. col is computed at line 9, which references Anext. All lines related to Anext should be kept. Line 3 is also included because Alast is involved in the comparison in line 8. The resulting code with only relevant statements is shown in Figure 3.b, which is much smaller than the original code. Accesses to w are replaced with calls to our runtime library, passing the index as a parameter.

Notice that, in some rare cases, a dependence slice function cannot be extracted, e.g. if a non-local array is involved in a conditional branch, such as “if(A[i]) A[i+1]=1”. However, such cases turn out to be rare.

#### 5. Computing Approximate Dependences in Parallel

Dependences can happen between any two iterations in the entire iteration space. Therefore, to get an accurate picture of the complete dependence graph, cross-iteration memory accesses must be studied with a global view. This is often expensive to compute at runtime, because either dependence computation must be serialized, or a large amount of data must be transmitted through inter-thread communications.

On the other hand, it is actually not necessary to obtain a full dependence graph for dependence-aware scheduling to be carried out efficiently. In fact, as long as an iteration can be scheduled to run ahead of the main thread, its execution time is deducted from the main thread, which results in a speedup. Therefore, whether an iteration can be moved 10 iterations ahead or 100 iterations ahead often makes no practical difference to the ultimate speedup of the given loop. Thus, in practice we can limit the scope of dependence checking to a smaller window of consecutive iterations, as long as the current iteration has enough room to be moved ahead. The general idea of adding more...
constraints by limiting the scope of dependence checking is applied to both dependence computation parallelization and approximate dependence computation on each thread.

5.1 Safety Rules

Since dependences are not computed accurately, it is crucial to make sure that the approximations do not lead to incorrect execution results. Note that, as long as dependences are computed conservatively, approximation only causes a more constrained scheduling of iterations, but doesn’t jeopardize the correctness of computation. Two safety rules are identified as guidelines for approximate computation of the dependences. Obeying these rules will prevent the violation of dependences.

**SAFETY RULE 1:** Even if there is no dependence from Iteration X to Iteration Y, we can safely assume the existence of such a dependence.

**SAFETY RULE 2:** Assuming that there is a dependence from Iteration X to Iteration Z, it is safe to assume the existence of a dependence from Iteration X to Iteration Y plus another dependence from Iteration Y to Iteration Z.

Safety rule 1 is obviously correct. For safety rule 2, assuming both dependences from Iteration X to Iteration Y and Iteration Y to Iteration Z transitively implies a dependence from Iteration X to Iteration Z. Therefore the original dependence is not violated.

5.2 Dependence Distance Array (DDA)

Some early work [9][6][15] builds a full dependence graph during dependence analysis. In general, a full dependence graph require \( O(N^2) \) memory space, where \( N \) is the number of iterations in a loop. The overhead is considerable when the number of iterations is large. Moreover, the cost of accessing, modifying the dependence graph also grows significantly for larger numbers of iterations.

Instead of building a full dependence graph, which effectively corresponds to a two-dimensional data structure, we adopt an alternative representation to flatten the graph into one dimension. This representation, named the Dependence Distance Array (DDA) is formally defined as follows.

**DEFINITION: Dependence Distance Array (DDA)**

For each iteration, we store one number in the Dependence Distance Array. For a given iteration, say \( Y \), this number corresponds to the distance (in terms of number of iterations) between Iteration \( Y \) and a predecessor Iteration of \( Y \), say \( X \), such that there is a dependence from \( X \) to \( Y \) and there are no dependences to \( Y \) from any of the iterations between \( X \) and \( Y \).

Dependence distance is a more constrained representation of the dependence graph. For example, a dependence distance number of 5 guarantees that the current iteration does not depend on any of the previous 4 iterations. By reducing the dependence information to a single distance number, it forces us to assume that there is a dependence between the current iteration and any of the previous iterations that are 5 iterations or more away from the current iteration, in this example.

For the example in Figure 1.a, Iteration 21 depends on Iterations 19 and thus has a dependence distance of 21-19=2. This dependence distances implies that Iteration 21 is dependent on Iterations 1 to 19, which is obviously a simplification as careful inspection of Figure 1 indicates that Iterations 6, 15, and 16 could be executed in parallel. Note that this approximation is safe according to safety rule 1, because more dependences are added but none are removed.

5.3 Parallelizing Dependence Computation

To take advantage of a multicore processor, dependence computation is parallelized. It is essential to have a completely parallel dependence computation algorithm, and it should be scalable to a large number of cores/threads.

To parallelize dependence computation, each thread is allocated a contiguous subset of the iteration space. Each thread only has knowledge about dependences among iterations that are allocated to the thread. Since a thread does not know about the dependences to the iterations not allocated to it, it has to assume the existence of dependences to all earlier iterations that are not on this thread in order to be safe.

5.4 Computation of Dependences on Each Thread

As mentioned earlier, data dependence analysis using large data structures to store all possible cross-iteration memory accesses [6][9][10][11][12][13][15] is too expensive. We studied two approaches to dependence computation on each thread. Both approaches attempt to limit memory cost by approximating the dependence relationship among iterations, but still make sure the scheduling is safe. They exclusively use thread private data during the dependence computation and do not involve any inter-thread communication.

5.4.1 Hash of Circular Buffers per Thread

This approach uses a hash set of circular buffers per thread. A circular buffer is a buffer for which entries are added sequentially, and where the buffer automatically overwrites old entries. A head pointer points to the newest entry. As shown in Figure 5.a, each address is hashed to get an index. Each index corresponds to a circular buffer, which stores all addresses mapped to it.

Each entry in the buffer contains three fields: (1) an address tag, (2) an iteration number, and (3) an access type. The address tag is used to distinguish among different addresses mapped to it.
the same circular buffer. The iteration number indicates the iteration ID where this access occurred. The access type distinguishes between read and write accesses.

Since a circular buffer discards old entries to accommodate new entries, it corresponds to a moving window that keeps track of only a fixed number of entries. Once an old entry is removed from the buffer, obviously, the information associated with that entry is lost. To minimally keep track of the discarded entries, each circular buffer is associated with a single additional value called last_iter. The last_iter value records the iteration number of the most recently discarded entry. When a search in the circular buffer does not yield a hit, the dependence is automatically assumed to be from last_iter. This again satisfies the safety rules.

Figure 5.b illustrates the inner workings of the circular buffer approach. Consider Variables A, B, and C, all hashed to the same dual-entry circular buffer. During Iteration 0, read accesses to Variables A and B are recorded in the circular buffer. The last_iter value is -1, indicating that no entry has been kicked out yet. During Iteration 1, a “read C” entry is added to the buffer by removing the oldest “read A at Iteration 0” entry from the buffer. The value of last_iter becomes 0, since Iteration 0 is the iteration number associated with the removed “read A at Iteration 0” entry. During Iteration 2, the write access to Variable B initiates a search in the circular buffer, resulting in a hit with the “read B at iteration 0” entry. This hit indicates a dependence from Iterations 0 to 2. Then, another entry is discarded when adding the “write B at Iteration 2” entry in the buffer. During Iteration 3, write access to Variable A triggers a search for A. This search misses; therefore, we conservatively assume a dependence from last_iter=0 to the current iteration.

In summary, the use of circular buffer instead of a conventional hash table helps achieve a constant memory requirement and a lower access latency. The number of past entries is fixed, but the window size (how many previous iterations examined) is not. This guarantees a low upper bound on execution time. In addition, it is possible to have two hash sets of circular buffers per threads, one for read and one for write accesses, instead of one hash set.

### 5.4.2 Hash of Bits Table per Thread

For sparse memory accesses, we propose an alternative approach that further reduces memory space and expedite dependence computation. This approach primarily uses a hash set of bits, referred here as the [hash set]. When the address hits in both the Thread and Iteration Table, we still assume no dependence, because finding an address in the hash set resolves the access location. When a search in the hash set is not successful, we use a separate set of read and write tables to distinguish between different dependence types (RAW, WAW, or WAR).

<table>
<thead>
<tr>
<th>Iterations</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accesses</td>
<td>rd A</td>
<td>rd C</td>
<td>wr B</td>
<td>wr A</td>
</tr>
<tr>
<td>Updated</td>
<td>rA0</td>
<td>rC1</td>
<td>rB0</td>
<td></td>
</tr>
<tr>
<td>Circular Buffer</td>
<td></td>
<td>rC1</td>
<td>rB2</td>
<td>wA3</td>
</tr>
<tr>
<td>Dep. Detected</td>
<td>last_iter=1</td>
<td>last_iter=0</td>
<td>last_iter=0</td>
<td>last_iter=1</td>
</tr>
<tr>
<td></td>
<td>Iter 2-&gt;0</td>
<td>Iter 2-&gt;0</td>
<td></td>
<td>Iter 3-&gt;0</td>
</tr>
</tbody>
</table>

**Figure 5. Illustration of the circular buffer.**
6. The Scheduler and Thread Communications

The scheduler synchronizes operations on other threads. It waits for the completion of dependence computation to start scheduling. A global array indicates the status of each iteration. Both the scheduler and the main thread access this array. For each iteration, there are three possible states: NO_EXEC, EXEC and BUSY. NO_EXEC means that the iteration has not been executed, whereas EXEC means it has been executed. BUSY means it is being executed. If the main thread or one of the worker threads takes the iteration to execute, it is marked as BUSY. Its state is changed to EXEC, once the execution ends.

The scheduler sequentially searches the Dependence Distance Array for iterations that can be executed earlier. It also monitors the progress of the main thread to make sure that the iteration to be scheduled on worker thread will not be executed by the main thread.

The scheduler can be implemented either as a number of functions that are run together with main-worker threads, or as a separate thread. Typically, using a separate thread is recommended for a lower overhead on the main thread.

7. Implementation and Other Considerations

The implementation was done as part of the IBM XL compiler. The XL compiler provides sophisticated loop optimizations, which include data analysis that identifies all independent loops. The approach proposed in this paper is built on top of existing optimizations with option –O5 turned on. In other words, all independent loops are excluded; optimizations such as privatization, loop fusion, unrolling etc. are performed as needed.

It involves two main components: (1) a component inside our optimizing compiler and (2) a runtime library that is linked with the generated code. On the compiler side, shown in Figure 7, we first manually identify target loops and outline these loops as individual functions. Due to the nature of such technologies, not all loops are good candidates. We then clone these individual functions such that they can be later converted to dependence slice functions. After all other loop parallelization optimizations have been applied, the compiler initiates a set of post-parallelization optimizations. We introduced a new one, labeled “Depaware Sched” to generate the dependence slice function and insert inlined calls to the runtime library.

A separate component is the runtime library, which is linked with the object code. The runtime library is composed of an entry function (the loop body is handed over as a function pointer), worker thread code, scheduler thread code and memory access handlers. Memory access handlers are a set of subroutines that can be invoked from the dependence slice function. Operations on the circular buffer or thread table are part of the memory access handler.

8. Experimental Results

8.1 Experimental Setup and Benchmarks

We carried out the experiments on a 16-core Power 5 1.5GHz machine with 32GB memory. Each core has two SMT threads, which provides a total of 32 hardware threads. In the default configuration, 16 worker threads are used. All of them are initially computing dependences. The scheduler runs on a separate thread.

As mentioned early, consecutive iterations are blocked to form larger ones. We limit the number of blocks to 1024. In other words, at most 1024 iteration blocks can be scheduled. A detailed list of configurations can be found in Table 2, including default
configuration for both circular buffer and thread table.

Table 3 shows all benchmarks and loops used in the experiments. We chose benchmarks from SPEC2000 (equake, art), SPEC2006 (lbm) and SEQUOIA Benchmarks [1] (lampsmk, irsmk and umtmk). SEQUOIA is provided by Lawrence Livermore National Laboratory to IBM for testing the performance of next-generation high performance supercomputers. Most benchmarks are for scientific computing, especially large simulation programs for chemical, physical and biological computations. Such benchmarks are computationally intensive, which is of great interest to loop optimizations. Given the overhead of thread initialization, we use large input sizes such that loops run for a sufficient amount of time. As shown in the last column of Table 3, all loops consist of a large number of iterations.

<table>
<thead>
<tr>
<th>Description</th>
<th>File</th>
<th>Line</th>
<th>#Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>art1</td>
<td>scanner.c</td>
<td>829</td>
<td>40M</td>
</tr>
<tr>
<td>equake1</td>
<td>quake.c</td>
<td>462</td>
<td>239M</td>
</tr>
<tr>
<td>equake2</td>
<td>quake.c</td>
<td>1795</td>
<td>229M</td>
</tr>
<tr>
<td>lammpsmk</td>
<td>main.c</td>
<td>122</td>
<td>4M</td>
</tr>
<tr>
<td>lbm1</td>
<td>lbm.c</td>
<td>128</td>
<td>33K</td>
</tr>
<tr>
<td>lbm2</td>
<td>lbm.c</td>
<td>186</td>
<td>10M</td>
</tr>
<tr>
<td>irsmk</td>
<td>rmatmult3.c</td>
<td>79</td>
<td>230</td>
</tr>
<tr>
<td>umtmk1</td>
<td>umtmk1.c</td>
<td>533</td>
<td>3.5M</td>
</tr>
<tr>
<td>umtmk2</td>
<td>umtmk2.c</td>
<td>564</td>
<td>264M</td>
</tr>
</tbody>
</table>

Table 3. Benchmarks and loops.

8.2 Results

First, we compare circular buffer, thread table and sequential execution in terms of execution time and dependence computation time. Figure 8 shows the results. For CB (Circular Buffer) and TT (Thread Table), each bar is split into two parts, the lower part is the dependence computation time, and the upper part is the execution time after dependences are computed. It is noteworthy that both schemes can dramatically cut down the total execution time. Compared with sequential execution, total execution time is cut to 39% for CB and 28% for TT. For either scheme, more time is spent on dependence computation than on the execution afterwards. It costs CB 22% of the original sequential execution time on dependence computation, while this number is 12% for TT, mainly because memory accesses are quite sparse for these scientific applications. The execution time after dependences are computed is close for CB and TT.

Translating to speedup (Figure 9), CB gets 3.1 times speedup; TT gets 3.9 times speedup. In generally, CB is slower in computing dependences due to its relatively larger data structure, which lengthens latency. TT is faster in the case of sparse accesses. If more dependences occur among iterations on the same thread, TT is likely to get worse performance. Notice that, for lammpsmk and irsmk, the performance of CB and TT is close. For irsmk, we observe that the scheduling time is very short. This can be seen from Figure 8 and the loop actually does not have real dependences, therefore using either CB or TT does not make much difference. For lammpsmk, there are many dependences, leading to a higher percentage of sequential execution during the time dependences are computed. Consequently, the speedup is much lower than other benchmarks, which also narrows the differences between the two approaches.

In Figure 11, the number of dependence computation threads is varied for CB. For all benchmarks except irsmk, allocating more dependence computation threads yields better performance. This can be explained as follows. With more dependence computation threads, dependence computation can be better parallelized, however the number of iterations allocated to each thread is less, this limits the scope of scheduling. For instance, if each thread is only given 5 iterations, each iteration
can only be moved ahead by 5 iterations, because iterations calculated on other threads are assumed to cause dependences. This is normally not an issue if enough iterations are allocated to each thread. However, for irsmk the total number of iterations is only 200. Only around 10 iterations are put on each thread, which greatly limits the scope of scheduling. On average, the speedups for 4 dep threads and 8 dep threads are 2.58 and 2.95 respectively.

![Figure 11. Speedups corresponding to different number of dependence computation threads for CB.](image)

In Figure 11, several different configurations of the circular buffer are tested. The number of circular buffers is varied from 16K to 512K. Each circular buffer may contain 2, 4, 8 or 16 entries. Apparently, more circular buffers with more entries lead to a better performance, but the performance improvement saturates around the default configuration, i.e. 256K x 8 entries.

![Figure 12. Speedups for different configurations of CB.](image)

Similar tests were done for TT as well. In Figure 13, it appears that the general trend is the same as CB, however the performance goes down more dramatically when less dep threads are supplied. This is perhaps because TT uses one table for each thread, as more iterations are allocated to each thread, more ambiguity occurs. On average, the speedups for 4 dep threads and 8 dep threads are 2.97 and 3.58 respectively.

![Figure 13. Speedups corresponding to different number of dependence computation threads for TT.](image)

As mentioned earlier, one of the main advantages of dependence-aware scheduling over prior schemes is that it performs very well in the worst case, i.e. when no parallelism is found, and the dependence computation cannot finish before the main thread finishes.

![Figure 14. Speedups for different configurations of TT.](image)

It is crucial to study the worst case, because dependence-aware scheduling is often not applicable to certain loops, e.g. small loops, loops with lots of dependences, etc. The compiler...
may not be able to fully identify such loops.

To test the worst case performance, we run dependence computation till the end of sequential execution. Even if dependence computation is finished, the thread loops back to the beginning so dependences are computed repeatedly. Continuously running worker threads stresses other cores and the memory subsystem. They also interfere with the main thread.

In Figure 15, data are shown for CB running with 4, 8 and 16 threads. It is a little surprising to see that, with 4 or 8 threads, the main thread actually gets a little speedup (1.6% for 4 threads, 0.9% for 8 threads. The slowdown is 1.2% for 16 threads. With more threads, cache contention is more severe. On the other hand, dependence computation also brings about a prefetching effect, which helps the main thread to shorten the latency of its memory accesses.

9. Related Work

Thread Level Speculation (TLS) [2][3][4][5] parallelizes code with potential memory access conflicts on a hardware platform that can detect those conflicts. Once a conflict is detected, some of the threads are rolled back to be re-executed. TLS does not require any help from the compiler side. However, it requires dedicated hardware support that is not currently available on commodity multicore. Instead, this work proposes a pure-software approach that can be deployed on any multi-threaded multicore machine immediately.

Two early work by Zhu and Yew [7] and Midkiff and Padua [8] proposed approaches that build wavefronts using simple strategies. Iterations are added to the current wavefront one by one if none of the data accessed in that iteration is accessed by any lower unassigned iterations. Computations of the wavefronts (i.e. dependence computation) and the actual work are interleaved. Polychronopoulos [14] extends their work by grouping contiguous iterations with no cross-iteration dependences. These approaches use $O(M)$ additional memory locations where $M$ is the size of the arrays tracked at runtime. They also use atomic synchronization primitives to detect dependences in parallel. Current multicore are quite sensitive to memory footprint and fine grain locking mechanisms. In addition to our lower memory footprint and synchronization-free parallel dependence computation for lower overheads, we also attempts to minimize the worst case slowdown by running the main thread sequentially regardless of dependence computations instead of mixing the two phases together.

In another extension of Zhu and Yew [7], Chen, Yew and Torrellas [10] proposed a two-phase approach that first builds a list of all accesses, then merges the information using a global Zhu and Yew algorithm. While this approach may lessen the computational and synchronization overhead compared to Zhu and Yew, it requires $O(NsR)$ additional memory, where $N$ is the number of iterations and $R$ is the number of static reference per iterations. Its dependence computation is also not synchronization free. Another extension by Krothapalli and Sadayappan [9] includes a parallel inspector that places all memory accesses in a dynamically allocated array and then sorts them according to the iteration number. Next, the inspector builds a dependence graph. Again, large data structure and computation-intensive operations are used, which could cause significant slowdown for loops with frequent cross-iteration accesses.

Saltz et. al. [11][13] handles loops without output dependence. A topological sort is performed to find the schedule. Leung [12] proposed a method to parallelize its inspector. Each processor first computes a local optimal schedule, then they are merged into a global one. Their approaches are efficient, however are limited due to the assumption that no output dependence is present.

Rauchwerger, Amato and Padua [6][15] proposed a scalable approach that parallelizes inspector, scheduler and executor. The inspector records all memory accesses locally then sorts them according to the iteration number. Per processor information is later used to form the dependence graph for each memory element. Finally, the scheduler generates wavefronts after cross-referencing data address information with iteration number. Although all phases are parallelized, due to the need to store all memory accesses and then to sort them, the worst-case overhead can be substantial.

The LRPD test was proposed by Lawrence Rauchwerger et. al. [16][17][19]. The underlying idea is to speculatively execute the loop in parallel and at the same time gather sufficient dependence information to subsequently test its legality. If the test fails, the loop is re-executed sequentially. The test performs well for a number of loops that are indeed parallel. The differences between LRPD and dependence-aware scheduling are two-fold. First, LRPD is mainly for fully parallel loops (i.e. either fully parallelized or sequential), although [18] extends it to partially parallel loops. Second, the goal of this work is to achieve a good worst-case performance, whereas LRPD incurs slowdown if the loop is not fully parallel.

Software TLS [20][21][22] is an alternative to hardware TLS. It simulates hardware TLS with software mechanisms. Typically, optimizations are performed to reduce the number of conflict detection instructions. Similar to hardware TLS, software TLS could suffer from frequent rollbacks due to conflicts. This makes the program run slower. Also, its execution time is not bounded in the worst case. Moreover, due to its heavy instrumentation, the amount of computation involved to process memory accesses and keep track of memory conflicts is prohibitive. In contrast, dependence-aware scheduling reduces checking overhead by using the slice function and aggressive approximations.

10. Conclusion

In conclusion, this paper proposes a runtime parallelization framework to parallelize hard-to-analyze loops. Because applications may have only limited amount of parallelism, our main goal is to reduce the negative impact of dependence computation to ensure that there is little slowdown even when there are no opportunities for parallelization. All dependence computations are fully parallel and synchronization-free. We have investigated multiple approaches, two of which we reported here, to reduce the memory footprint of dependence computation, increase its speed through safe approximations, and alleviate any interference with the main thread.

Our results indicate up to 4x speedup, and almost no slowdown even if dependence computation is unused. We expect such technology to be more popular for future heterogeneous multicore processors, where the main thread can be better isolated from its worker threads for an even more steady performance.

ACKNOWLEDGMENTS
This work has benefited from other researchers in IBM T. J. Watson Research Laboratory including Zehra Sura, Lakshminarayanan Renganarayana, and Tong Chen.

REFERENCES


