Uncovering Hidden Loop Level Parallelism in Sequential Applications

Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{hongtaoz,mehrara,lieberm,mahlke}@umich.edu

Abstract

As multicore systems become the dominant mainstream computing technology, one of the most difficult challenges the industry faces is the software. Applications with large amounts of explicit thread-level parallelism naturally scale performance with the number of cores, but single-threaded applications realize little to no gains with additional cores. One solution to this problem is automatic parallelization that frees the programmer from the difficult task of parallel programming and offers hope for handling the vast amount of legacy single-threaded software. There is a long history of automatic parallelization for scientific applications, but the techniques have generally failed in the context of general-purpose software. Thread-level speculation overcomes the problem of memory dependence analysis by speculating unlikely dependences that serialize execution. However, this approach has lead to only modest performance gains. In this paper, we take another look at exploiting loop-level parallelism in single-threaded applications. We show that substantial amounts of loop-level parallelism is available in general-purpose applications, but it lurks beneath the surface and is often obfuscated by a small number of data and control dependences. We adapt and extend several code transformations from the instruction-level and scientific parallelization communities to uncover the hidden parallelism. Our results show that 61% of the dynamic execution of studied benchmarks can be parallelized with our techniques compared to 27% using traditional thread-level speculation techniques, resulting in a speedup of 1.84 on a four core system compared to 1.41 without transformations.

1 Introduction

Due to power dissipation and design complexity issues of building faster uniprocessors, multicore systems have emerged as the dominant architecture for mainstream computer systems. Semiconductor companies now put two to eight cores on a chip, and this number is expected to continue growing. One of the most difficult challenges going forward is software: if the number of devices per chip continues to grow with Moore's law, can the available hardware resources be converted into meaningful application performance gains? In some regards, the embedded and domain-specific communities have pulled ahead of the general-purpose world in taking advantage of available parallelism, as most system-on-chip designs have consisted of multiple processors and hardware accelerators for some time. However, these system-on-chip designs are often deployed for limited application spaces, requiring hand-generated assembly and tedious programming models. The lack of necessary compiler technology is increasingly apparent as the push to run general-purpose software on multicore platforms is required.

In the scientific community, there is a long history of successful parallelization efforts [1, 3, 7, 12, 15]. These techniques target counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Loop-level and single-instruction multiple-data parallelism are extracted to execute multiple loop iterations or process multiple data items in parallel. Unfortunately, these techniques do not often translate well to general-purpose applications. These applications are much more complex than those typically seen in the scientific computing domain, often utilizing pointers, recursive data structures, dynamic memory allocation, frequent branches, small function bodies, and loops with small bodies and low trip count. More sophisticated memory dependence analysis, such as points-to analysis [23], can help, but parallelization often fails due to a small number of unresolvable memory accesses.

Explicit parallel programming is one potential solution to the problem, but it is not a panacea. These systems may burden the programmer with implementation details and can severely restrict productivity and creativity. In particular, getting performance for a parallel application on a heterogeneous hardware platform, such as the Cell architecture, often requires substantial tuning, a deep knowledge of the underlying hardware, and the use of special libraries. Further, there is a large body of legacy sequential code that cannot be parallelized at the source level.

A well-researched direction for parallelizing general-purpose applications is thread-level speculation (TLS). With TLS, the architecture allows optimistic execution of code regions before all values are known [2, 13, 14, 25, 30, 32, 34, 39]. Hardware structures track register and memory accesses to determine if any dependence violations occur. In such cases, register and memory state are rolled back to a previous correct state and sequential re-execution is initiated. With TLS, the programmer or compiler can delineate regions of code believed to be independent [4, 9, 18, 20]. Profile data is often utilized to identify regions of code that are likely independent, and thus good candidates for TLS.

Previous work on TLS has yielded only modest performance gains on general-purpose applications. The POSH compiler is an excellent example where loop-based TLS yielded approximately 1.2x for a 4-way CMP and loop combined with subroutine TLS yielded approximately 1.3x on SPECint2000 benchmarks [18]. That result improves upon prior results reported for general-purpose applications by the Stampede and Hydra groups [13, 32]. One major limitation of prior work is that parallelization is attempted on unmodified code generated by the compiler. Real dependences (con-

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

trol, register, or memory) often mask potential parallelism. A simple example is the use of a scalar reduction variable in a loop. All iterations update the reduction variable, hence they cannot be run in parallel. One notable exception is the work by Prabhu and Olukotun that looked at manually exposing thread-level parallelism (TLP) in Spec2000 applications [26]. They examined manually transforming applications to expose more TLP, including introducing parallel reductions. They showed substantial increases in TLP were possible using a variety of transformations for traditional sequential applications. However, many of the transformations were quite complex, requiring programmer involvement.

Our work is directly motivated by Prabhu and Olukotun. We examine the feasibility of automatic compiler transformations to expose more TLP in general-purpose applications. We target automatic extraction of loop-level parallelism, where loops with sets of completely independent loop iterations, or DOALL loops, are identified, transformed, and parallelized. Memory dependence profiling is used to gather statistics on memory dependence patterns in all loop bodies, similar to prior work [18]. Note that we are not parallelizing inherently sequential algorithms. Rather, we focus on uncovering hidden parallelism in implicitly parallel code.

We examine the use of TLS as a method to overcome the limitations of static compiler analysis. This is the same conclusion reached by prior work. However, we look beyond the nominal code generated by the compiler to find parallel loops. We show that substantial loop-level parallelism lurks below the surface, but it is obstructed by a variety of control, register and memory dependences. To overcome these dependences, we introduce a novel framework and adapt and extend several code transformations from domains of instruction-level parallelism and parallelization of scientific codes. Specifically, our contributions are:

- DOALL loop code generation framework - We introduce a novel framework for speculative partitioning of chunked loop iterations across multiple cores. The template handles complex cases of uncounted loops as well as counted ones and takes care of all scalar live-outs.
- TLP-enhancing code transformations - We design several code transformations to break cross iteration dependences in nearly DOALL loops. These transformations are not entirely new, but rather are variants of prior techniques that have different objectives and are adapted to work in the presence of uncertain dependence information and TLS hardware. The optimizations consist of of speculative loop fission, speculative prematerialization, and isolation of infrequent dependences.

2 Parallelization Challenges

To illustrate more concretely the challenges of identifying TLP using compiler analysis in general-purpose code, we examine the frequency of provable DOALL loops in a variety of applications. We use the OpenImpact compiler system that performs memory dependence analysis using interprocedural points-to analysis [23]. A total of 41 applications (all C source code, or C code generated from f2c) from four domains are investigated: SPECfp, SPECint, MediaBench, and Unix utilities. The lower portion of the bars in Figure 1 show the ability of an advanced compiler to expose provable DOALL parallelism. For each application, the fraction of sequential execution that is spent in provable DOALL loops is presented. To derive this value, the execution frequency of all

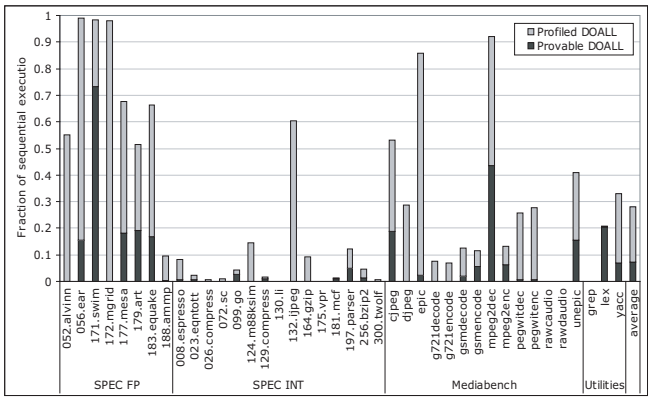

Figure 1: Fraction of sequential execution covered by provable DOALL loops identified through compiler analysis (bottom portion of bars) and speculative DOALL loops identified through profiling (top portion of bars).

static operations that reside in at least one provable DOALL loop are summed and divided by the total dynamic operation count. The figure shows that the compiler analysis is not very successful with the exception of two cases, 171.swim from SPECfp and mpeg2dec from MediaBench. The compiler is generally most successful in SPECfp, where previous scientific parallelization techniques are applicable. However, the pointer analysis is only partially successful at resolving memory dependences in the non-scientific code.

TLS has the potential to provide large performance gains by allowing speculative parallelization of loops where compiler analysis alone is unsuccessful. One key issue for TLS is to parallelize the loops that have low probability memory dependences. Memory profiling is a way to estimate the memory dependences in a program. The memory profiler runs the application on a sample input and records the memory address accessed by every load and store. If two memory instructions access the same location, a memory dependence is recorded.

For the purpose of parallelizing loops, we only care about cross iteration dependences. If the loops are nested, we need to know in what nesting level each memory dependence is happening. Furthermore, when the loop contains function calls, we want to know if the called function accesses any global variable which causes cross iteration memory dependence. Therefore, we developed a control-aware memory profiler to identify speculative DOALL loops.

The upper portion of the bars in Figure 1 show the fraction of serial runtime spent in DOALL loops identified by the profiler. A loop is speculative DOALL if it contains zero or very few cross iteration memory dependences, and it contains no cross iteration register and control dependences. Note that simple register and control dependences, such as those cause by induction variables, can easily be eliminated and hence are ignored in this analysis. Comparing the profiled and provable results, we see that many more DOALL loops are identified using profile information than using compiler memory analysis. On average, 28% of the sequential execution is contained within speculative DOALLs, compared with 8% in provable DOALLs. As expected, the profiler is highly effective with the SPECfp applications, which are known to contain large amounts of loop-level parallelism. However, for the remaining applications, the results are generally disappointing. With the exception of a few media ap-

plications, most of these applications contain few DOALL loops.

These poor results were confusing as we had observed that many loops in these applications contained no statistically significant cross-iteration memory dependences. Hence, if we were only looking at memory dependences, the number of speculative DOALLs would be much larger. The problem is that the loops are not DOALL due to other dependences - namely cross iteration register and control dependences. If these dependences could be broken by the compiler, then the number of DOALL loops would increase substantially. The remainder of the paper explores this direction of research, namely a set of compiler transformations to break these dependences within the context of a speculative execution environment.

3 Architectural Support

Speculative DOALL loops require several underlying features to ensure correct execution and make recovery actions when a speculation violation occurs. These features can be implemented in hardware, software, or a combination of the two. Traditional TLS hardware can support speculative execution of all types of loops and acyclic code [30, 32, 13]. Since we are only executing speculative DOALL loops, a subset of TLS hardware is required. Furthermore, since many of the loops studied have small bodies and low trip counts, the overhead of parallel execution must be minimized. In this section, we specify the underlying hardware model and assumptions about its operation.

Our target architecture is a standard chip multiprocessor with coherent L1 caches and a shared L2 cache. The system is extended with three major features to support speculative DOALL execution. First, a transactional memory system similar to LogTM [22, 37] is utilized to detect memory dependence violations and rollback execution if necessary. We assume ordered transactions. In the case of a conflict, the transaction with the higher ID is aborted. Larger transaction IDs are assigned to higher groups of loop iterations to maintain sequential semantics. Since we only parallelize loops with statistically zero dependences, memory values are not forwarded from previous transactions to later transactions to simplify the hardware implementation. If a later transaction uses values stored in a previous transaction, the later transaction is aborted. Several new instructions are added to the instruction set to expose the transactional memory to the compiler. The XBEGIN instruction marks the beginning of a transaction. XBEGIN takes the address of the abort handler as an operand. The XCOMMIT instruction marks the end of a transaction and commits the speculative state.

Second, a scalar operand network, similar to that in the Raw architecture [33], is used to communicate register values between cores. A mechanism similar to register channels [11, 10] or a synchronization array [28] can also be used to support register communication and synchronization. Each core is extended with send and receive queues and simple routing logic (XY routing is assumed). Two new instructions, SEND and RECV, are added to the instruction set. The SEND instruction has two source operands, a register and a destination core ID. It reads the value in the source register and sends it to the destination core. The RECV instruction also takes two source operands, a target register and a sender core ID. When a RECV is executed, it looks in the incoming message queue in the core for messages from the sender ID. If such a message is found, it moves the value to the target register; otherwise, it stalls the core and waits for the message.

SEND and RECV operations also provide an efficient mechanism to guarantee the ordering of instructions in different cores. We commit loop iterations in original program order to maintain sequential semantics by passing a commit permission token between the cores using SEND/RECV instructions.

Finally, the hardware supports low latency thread spawning to efficiently exploit parallelism in small loops. We assume the operating system pre-allocates several cores to each application. To spawn a new thread, a core simply sends a program counter (PC) value to another core to initiate its execution. Since the compiler explicitly controls thread spawning, the live-in scalar values for the slave are explicitly passed from the master using the scalar operand network.

4 Uncovering Hidden Loop Level Parallelism

As shown in the previous sections, without any code transformation, out-of-the-box loop level parallelization opportunities for general applications are limited, even with TLS hardware support. After manually studying a wide range of loops, we found that many parallel opportunities were hidden beneath the seemingly sequential code. With proper code transformations, critical cross iteration dependences can be untangled resulting in many more speculative DOALL loops. In this section, we first introduce our code generation scheme for speculative DOALL loops. It handles both counted loop and uncounted loop with cross iteration control dependences. Subsequently, we present techniques to handle cross iteration dependences that hinder loop level parallelism. In addition to some well-known techniques, we introduce three novel transformations to untangle register and memory dependences: speculative loop fission, speculative prematerialization, and isolation of infrequent dependences.

4.1 Code Generation

After choosing candidate loops for parallelization using the profile information, the compiler distributes loop execution across multiple cores. In this work, we categorize DOALL loops into DOALL-counted and DOALL-uncounted. In DOALL-counted, the trip count is known when the loop is invoked (note, the trip count is not necessarily a compile-time constant). However, for DOALL-uncounted, the trip count is unknown until the loop is fully executed. *While* loops and *for* loops with conditional break statements are two examples of DOALL-uncounted loops that occur frequently. In these cases, the execution of every iteration depends on the outcome of exit branches in previous iterations. Therefore, these loops contain cross iteration control dependences. In this section, we introduce our code generation framework for handling control dependences and executing both speculative DOALL-counted and DOALL-uncounted loops.

Figure 2 shows the detailed implementation of our code generation framework. In the proposed scheme, the loop iterations are divided into chunks. The operating system passes the number of available cores and the chunk size to the application. Our framework is flexible enough to use any number of available cores for loop execution. We insert an outer loop around the original loop body to manage the parallel execution between different chunks. Following is a description of the functionality of each segment in Figure 2.

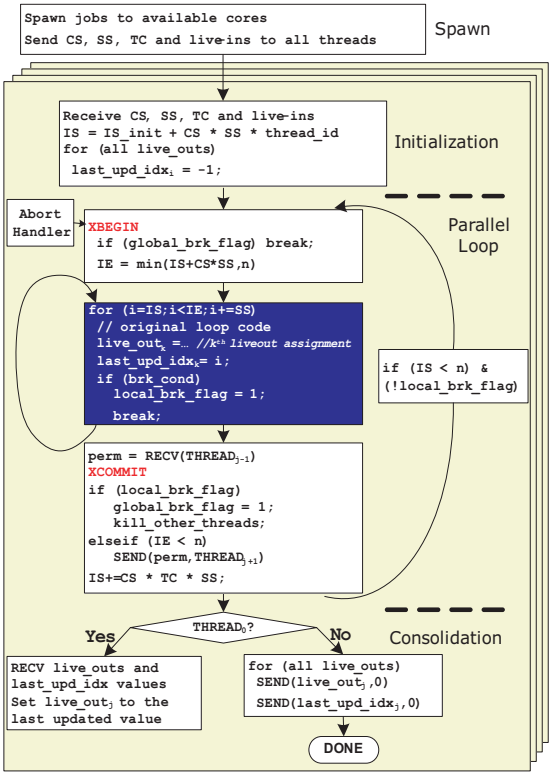

Figure 2: Detailed code generation framework for speculative loop parallelization. Transaction scope is marked by XBEGIN and XCOMMIT. (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count)

Spawn: To start, the master thread spawns threads containing chunks of loop iterations. It sends the necessary parameters (chunk size, thread count, etc.) and live-in values to all threads.

Initialization: In the initialization block, all participating cores receive the required parameters and live-in values. Since live-in values are not changed in the loop[1], we only send them once for each loop execution. This block also computes the starting iteration, IS, for the first chunk. After initialization, each core manages its own iteration start value, thus parallel execution continues with minimum interaction among the threads.

In order to capture the correct live-out registers after parallel loop execution, we use a set of registers called *last-upd-idx*, one for each conditional live-out (i.e., updated in an if-statement). When a conditional live-out register is updated, we set the corresponding *last-upd-idx* to the current iteration number to keep track of the latest modifications to the live-out values. If the live-out register is unconditional (i.e., updated in every iteration), the final live-out value can be retrieved from the last iteration and no *last-upd-idx* is needed.

Abort Handling: The abort handler is called when a transaction aborts. If the TM hardware does not backup the register file, we can use the abort handler to recover certain register values in case of transaction abort. More specifically, we need to recover the live-out and *last-upd-idx* register values. We need to backup these registers at the beginning of each transaction, and move the backup value to the registers in the abort handler. Moreover, we also add recovery code in the abort handler for some of our transformations as described in the next section.

Parallel Loop: The program stays in the parallel loop segment as long as there are some iterations to run and no break has happened. In this segment, each thread executes a set of chunks. Each chunk runs iterations from IS to IE. The value of IS and IE are updated after each chunk using the chunk size (CS), thread count (TC) and step size (SS).

Each chunk of iterations are enclosed in a transaction, demarcated by XBEGIN and XCOMMIT instructions. The transactional memory monitors the memory accessed by each thread. It aborts the transaction running higher iterations if a conflict is detected, and restarts the transaction from the abort handler. Chunks are forced to commit in-order to maintain correct execution and enable partial loop rollback and recovery. In order to minimize the required bookkeeping for this task, we use a distributed commit ordering technique in which each core sends commit permission to the next core after it commits successfully. These permissions are sent in the form of dummy values to the next core. The $RECV$ command near the end of the main block causes the core to stall and wait for dummy values from the previous core.

For uncounted loops, if a break happens in any thread, we don't want to abort higher transactions immediately, because the execution of the thread is speculative and the break could be a false break. Therefore, we use a local variable *local_brk_flag* in each thread to keep track of if a chunk breaks. If the transaction commits successfully with *local_brk_flag* set, the break is not speculative any more, and a transaction abort signal is sent to all threads using a software or hardware interrupt mechanism. In addition, a *global_brk_flag* is set, so that all threads break the outer loop after restarting the transaction as a result of the abort signal. The reason for explicitly aborting higher iterations is that if an iteration is started by misspeculation after the loop breaks, it could produce an illegal state. The execution of this iteration might cause unwanted exceptions or might never finish if it contains inner loops.

Consolidation: After all cores are done with the execution of iteration chunks, they enter the consolidation phase. In this period, each core sends its live-outs and *last-upd-idx* array to $THREAD_0$ that selects the last updated live-out values. All threads are terminated after consolidation and $THREAD_0$ continues with the rest of program execution.

We carefully designed the framework to keep most of the extra code outside the loop body, so they only execute once per chunk . The overhead in terms of total dynamic instructions is quite small.

4.2 Dependence Breaking Optimizations

This section focuses on breaking cross iteration register dependences that occur when a scalar variable is defined in one iteration and used in another. First, we examine several traditional techniques that are commonly used by parallelizing compilers for scientific applications: variable privatization, reduction variable expansion, and ignoring long distance memory dependences. These optimizations are adapted to a speculative environment. Then, we propose three optimizations specifically designed for a speculative environment: speculative loop fission, speculative prematerialization and infrequent dependence isolation. These transformations are adaptations of existing ones used in the

[1]If a live-in value is changed in the loop, it generates a cross iteration register dependence and the loop cannot be parallelized.

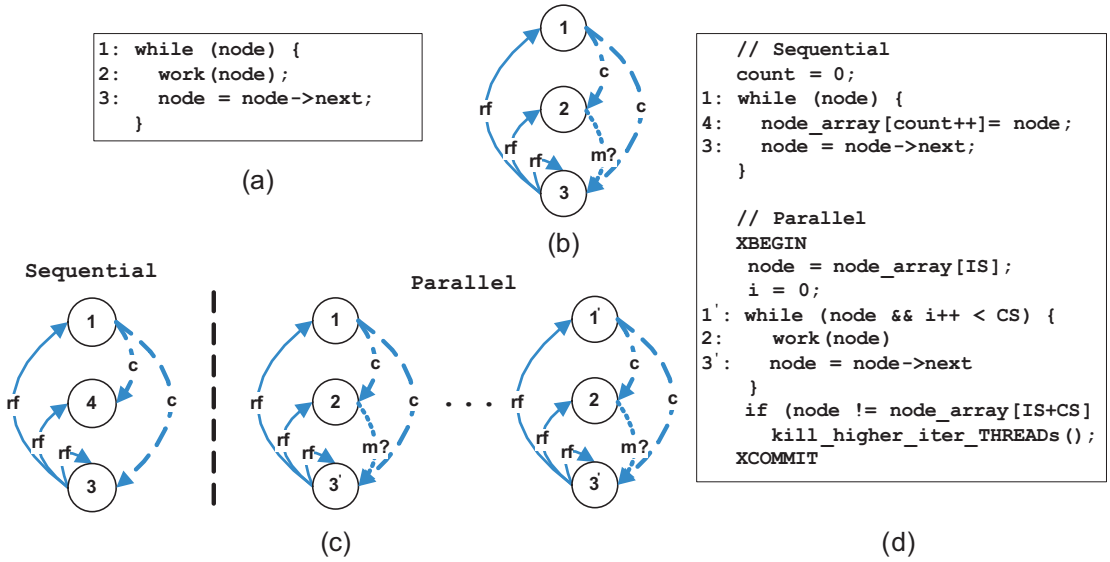

Figure 3: Speculative loop fission (a) Original loop (b) Original data flow graph (c) Data flow graph after fission (d) Loop after fission - (rf: register flow dependence, c: control dependence, m?: unlikely cross-iteration memory dependence

scientific and ILP compilers.

4.2.1 Traditional Dependence Breaking Optimizations

Variable privatization. Cross iteration input, anti- and output dependences can be removed by register privatization. Since each core has a separate register file, register accesses in different cores are naturally privatized. Live-in scalars are broadcast to each core during initialization, thus all false dependences on scalars are removed. Handling output dependences for live-out variables is tricky. Since the value of a live-out variable is used outside the loop, we need to find out which core performed the last write to the register. This is not obvious if the register is updated conditionally. As described in the previous section, the code generation template handles this case by assigning an integer value on each core for each live-out register. This value is set to the last iteration index where the live-out variable is written. The compiler inserts code after the loop (in the consolidation block in Figure 2) to set the live-out registers to their last updated values in the loop based on the stored iteration index.

Reduction variable expansion. Reduction variables, such as accumulators or variables that are used to find a maximum or minimum value, cause cross iteration flow dependences. The most common case is the *sum* variable when all elements of an array are summed. These dependences can be removed by creating a local accumulator (or min/max variable) for each core, and privately accumulating the totals on each core. After the loop and in the consolidation block, local accumulators are summed or the global min/max is found amongst the local min/max's.

Ignoring long distance memory dependences. When the number of iterations between two memory dependences is larger than some threshold, there is an opportunity for parallelization by simply ignoring the dependence. Intuitively, if the distance between memory accesses is n, the compiler can make $n-1$ iterations execute in parallel. Subsequently, if we set the chunk size as $cross\_iteration\_distance/number\_of\_cores$, our scheme in Section 4.1 generates the proper code for parallel execution of the loop.

4.2.2 Speculative Loop Fission

By studying benchmarks, we observed that many loops contain large amounts of parallel computation, but they cannot be parallelized due to a few instructions that form cross iteration dependence cycles. We call these loops *almost_DOALL* loop as the bulk of the loop is DOALL, but a small recurrence cycle(s) inhibits parallelization. The objective of speculative loop fission is to split the almost_DOALL into two parts: a sequential portion that is run on one core followed by a speculative DOALL loop that can be run on multiple cores. The basic principles of this optimization are derived from traditional loop fission or distribution [1].

Figure 3(a) shows a classic example of such a loop. A linked list is iterated through, with each iteration doing some work on the current node. Figure 3(b) shows the data dependence graph for the loop, with the important recurrence of operation 3 to itself. Note that there may be an unlikely memory dependence between operations 2 and 3 as indicated by the "m?" edge in the graph. Such a situation occurs when the compiler analysis cannot prove that the linked list is unmodified by the work function. For this loop, the sequential portion consists of the pointer chasing portion (i.e., operation 3) and the DOALL portion consists of the work performed on each node (i.e., operation 2).

The basic transformation is illustrated in Figure 3(c). The strongly connected components, or SCCs, are first identified to compose the sequential portion of the loop. Dependences that will be subsequently eliminated are ignored during this process. These include control dependences handled by the DOALL-uncounted schema (i.e., the control dependence from operation 1 to 3), unlikely memory dependences (i.e., the memory dependence from operation 2 to 3), and register dependences caused by reduction variables. In this example, the SCC is operation 3. The sequential portion is then populated with two sets of nodes. First, copies of all

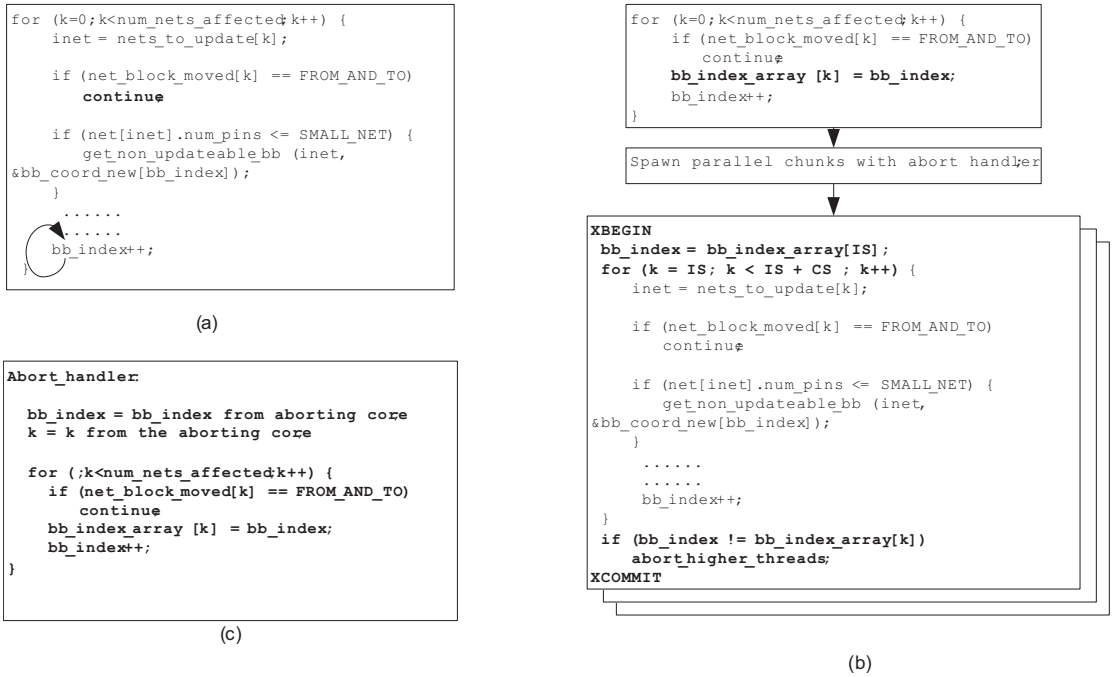

Figure 4: Example of speculative loop fission from 175.vpr: (a) original loop (b) loop after fission (c) abort handler.

dependence predecessors of the SCC are added (operation 1 in Figure 3(c)). Second, a new operation is introduced for each register flow edge that is cut between the SCCs and the remaining operations. In the example, there is a register flow edge from operation 3 to 2. A new operation is created (operation 4) that stores the value communicated via the edge into an array element. For this example, each node pointer is stored into the array. In essence, the register flow dependence is converted into a through-memory dependence. The result is the dependence graph shown on the left portion of Figure 3(c) and the code shown at the top of Figure 3(d).

The parallel portion of the loop consists of the entire original loop, including the SCCs, with a few modifications as shown in Figures 3(c) (right portion) and 3(d) (bottom portion). Each parallel chunk is seeded with a single value computed in the sequential loop for each register flow edge that was cut. In the example, *node* is set to *node_array[IS]* or the index of the starting iteration of the chunk. The body of the DOALL is identical to the original loop, except that only a fixed number of iterations are performed, CS or chunk size. Note that each parallel chunk is sequential, yet all parallel chunks are completely decoupled due to array variables produced by the sequential loop (i.e., *node_array*).

The final change is a test to ensure that each live-out SCC variable has the same value that was computed in the sequential loop. For the linked list example, this tests whether the current parallel chunk modified the starting element of the next chunk. This test combined with the transaction commit ensures that the linked list was not modified during the parallel portion. In cases where the compiler can prove no modifications are possible, this check is not necessary. The final parallel code is presented in Figure 3(d). Note that only the transaction scope portion of the code is shown for clarity. This code is dropped into the DOALL-uncounted template in Figure 2 to complete the parallelization.

Our loop fission scheme is different from traditional loop fission technique in that both the sequential and parallel loops are speculative. Since the sequential loop contains computations from every iteration, it could conflict with one or more of the parallel chunks. For example, in Figure 3(a), the work function could modify the linked list. This means that *node_array* contains one or more incorrect values. Such a memory dependence violation must be detected and rollback performed. The combination of the transactional semantics and the additional tests added after each parallel chunk to test the SCC variables ensure there are no unexpected memory dependences between two parallel chunks (transaction commit) and between the sequential and parallel chunks (explicit test inserted by the compiler). To simplify the problem, we don't allow inclusion of any store instruction in the sequential loop besides the ones that write to the new arrays. Our experiments show that very few fission candidates are lost by this requirement. When a parallel loop chunk reaches the end of its execution, it can commit only if all previous chunks have committed and no conflicts are detected, thereby ensuring correctness.

When the abort handler is invoked due to a memory dependence conflict, it must first abort all threads executing higher numbered iterations. Then, it restarts the execution of the sequential loop to re-initialize all the relevant values for the new arrays (i.e., *node_array* in the example). To ensure that modifications to data structures by later iterations do not affect earlier iterations, the sequential loop is run only from the starting iteration of the next thread after abort (iteration start + chunk size or IS+CS) to reset only the relevant portion of the new array(s).

To show a real example of speculative loop fission, Figure 4(a) presents an important $almost\_DOALL$ loop from the SPECint application 175.vpr. This example is different from the previous example in that it is not a linked list traversal. The variable *bb_index* carries a cross iteration register dependence. The variable is not an induction variable be-

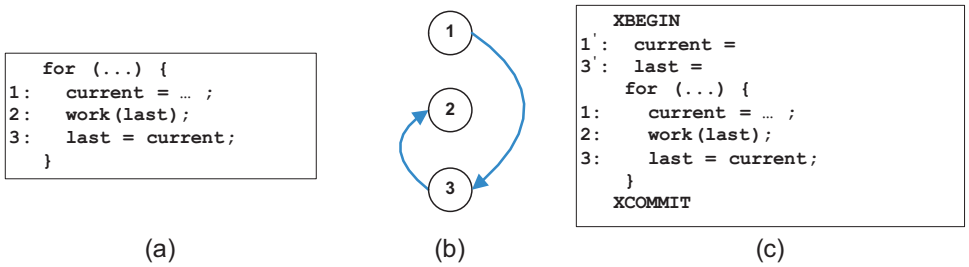

Figure 5: Example of speculative prematerialization: (a) original loop, (b) dataflow graph, and (c) loop after transformation.

cause it is not updated in every iteration due to the *continue* statement. The split loops are shown in Figure 4(b). The first loop is the sequential loop and contains the cross iteration dependences. It produces value of *bb_index* on every iteration and stores them to a new array called *bb_index_array*. The second loop is the parallel loop, where each chunk is decoupled through the use of *bb_index_array*. Finally, the abort handler for the loop is presented in Figure 4(c).

Two alternatives for parallelizing almost_DOALL loops are DOACROSS [1] and speculative decoupled software pipelining (DSWP) [35]. We consider speculative fission a better option than DOACROSS for two reasons. First, DOACROSS does not work with iteration chunking. If chunks of many iterations are executed in the DOACROSS manner, the first iteration in a chunk has to wait for data from the last iteration of the previous chunk, which basically sequentializes execution. For loops with small bodies, iteration chunking is very important to get performance improvement. Second, DOACROSS execution is very sensitive to the communication latency between threads because each iteration has to wait for data from the previous iteration. With speculative loop fission, the communication between the sequential part and the parallel part can happen in parallel and thereby the total execution time would be much shorter.

Speculative DSWP converts almost_DOALL loops into a producer-consumer pipeline. This has the advantage of overlapping the sequential and parallel portions. However, when the two portions are not relatively equal sized, the pipeline can be unbalanced. This problem can be alleviated by replicating pipeline stages. We believe DOALL execution is more scalable and more compatible with conventional transactional semantics.

4.2.3 Speculative Prematerialization

A special type of cross iteration register dependence can be removed through a transformation called speculative prematerialization. The idea of prematerialization is to execute a small piece of code before each chunk to calculate the live-in register values, so the chunks can be executed in parallel instead of waiting for all previous iterations to finish. Rematerialization is a technique commonly used by register allocators where its more efficient to recompute a value than store it in a register for a long period of time. Here the objective is different, but the process is similar.

Prematerialization can remove cross iteration dependences on registers that are not part of dependence cycles and are defined in every iteration. For each register that satisfies those two conditions, pre-execution of at most one iteration will generate the live-in value for a chunk. If a loop contains n registers that need to be prematerialized, at most n iterations need to be pre-executed.

Figure 5 illustrates the transformation. The original loop and dataflow graph are presented in Figures 5 (a) and (b). There is a cross iteration register flow edge from operation 3 to 2 corresponding the the variable *last*. The transformation is accomplished by peeling off a copy of the source of the register flow dependence and all its dependence predecessors. In this case, the source of the register flow dependence (operation 3) and its dependence predecessor (operation 1) are peeled and placed in the loop preheader. The resultant loop is shown in Figure 5(c). On the surface, this loop is still sequential as the dependence between operations 3 and 2 has not been eliminated. However, the peeling decouples each chunk from the prior chunk allowing the chunks to execute in parallel.

One important thing to note is that the prematerialization code is speculative because other iterations could modify variables it uses. This is akin to speculative fission where the linked list is modified during its traversal. In the simple example, a pointer used to compute *current* could be changed, thereby invalidating the prematerialized variables. Thus, the prematerialization code must be part of the transaction that contains the chunk. If any memory conflict is detected in the prematerialization code or the loop itself, the transaction corresponding to higher number iterations is aborted and restarted.

To illustrate a real application of prematerialization, Figure 6 shows a loop in the application djpeg from MediaBench. The variables *lastcolsum*, *thiscolsum*, and *nextcolsum* form a 3-wide sliding window in the loop. Variables *nextcolsum* and *lastcolsum* both carry cross iteration dependences that prevent DOALL execution. Speculative prematerialization can be applied because the value of *nextcolsum* and *lastcolsum* are defined in every iteration, and the cross iteration dependences do not form cycles. The right half of Figure 6 shows the parallel code after prematerialization. A prematerialization block is inserted before each chunk to compute the live-in values for *nextcolsum* and *lastcolsum*. In the prematerialization code, portions of the previous two iterations are executed to prematerialize two variables.

4.2.4 Infrequent Dependence Isolation

Another form of almost_DOALL loops are loops with infrequently occurring cross-iteration dependences. The sources or sinks of the dependence edges are contained in infrequently executed conditional clauses. Thus, the majority of the time the loops are DOALL. Isolation does not break any dependences, but rather transforms the control flow structure of the code to allow the compiler to parallelize the portion of the loop that is DOALL. The transformation is sim-

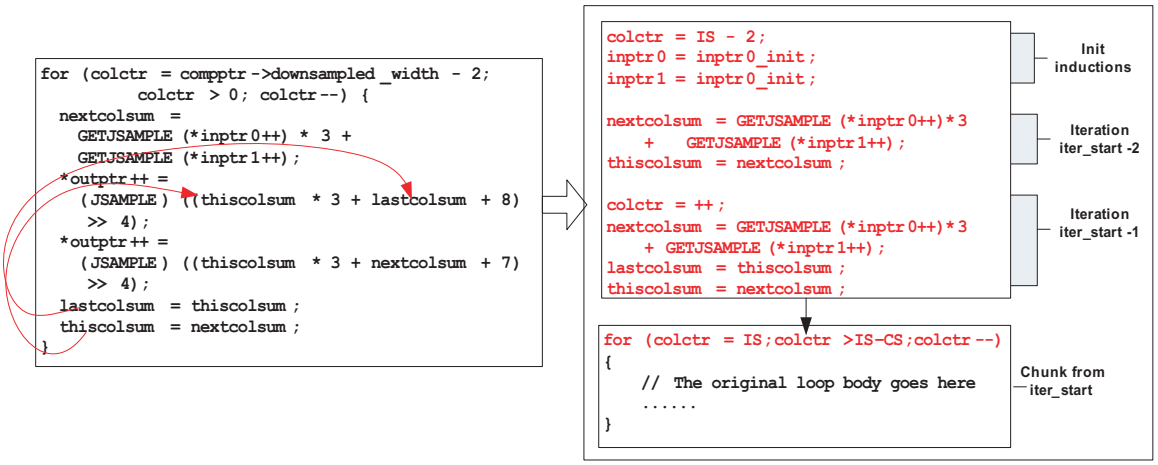

Figure 6: Example of speculative prematerialization from djpeg.

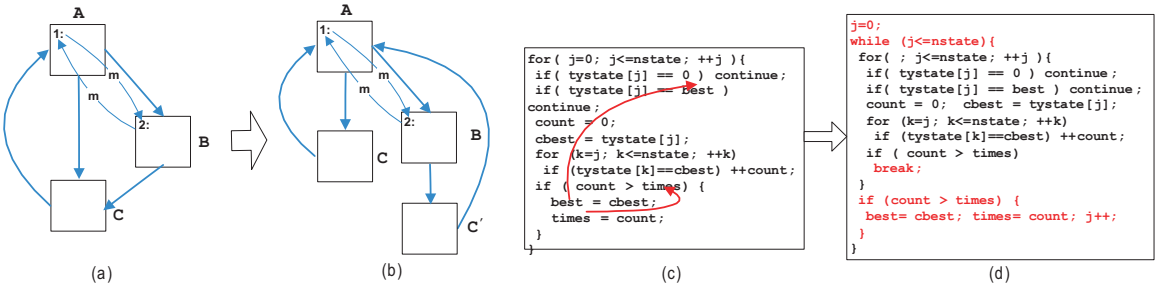

Figure 7: Example of dependence isolation: (a) mechanics of transformation for an abstract loop, (b) example loop from yacc.

ilar to hyperblock formation, but again the objectives are changed [19].

Cross-iteration register and memory dependences are eligible for isolation as well as calls to libraries where the compiler does not have complete memory dependence information. Library calls are typically treated conservatively and and thus inhibit parallelization. Isolation optimizes the common case by restructuring a loop into a nested loop. The schematic transformation is illustrated in Figure 7(a). The example loop consists of three basic blocks, A, B, and C. A dependence cycle exists between operations 1 and 2, contained in blocks A and B, respectively. Assume that block B is infrequently executed. Isolation converts the A-B-C loop into a nested loop structure as shown in the figure. The inner loop contains the frequent, DOALL portion, namely A-C. And, the outer loop contains the sequential portion, namely A-B-C. Block C is duplicated as in hyperblock formation to eliminate side entrances into the loop. The resultant inner loop is an DOALL-uncounted. When control enters block B, parallel execution is aborted, and the cross iteration dependence is properly enforced.

Figure 7(b) illustrates the application of dependence isolation to a loop from the Unix utility yacc. The left hand portion of part (b) shows the original code in which the outer *for* loop is not parallellizable due to two cross-iteration register dependences that are shown by arrows. However, according to the profile information, the *if* statement at the bottom of the loop rarely evaluates to True. Therefore, we can transform the loop to that in the right hand portion of Figure 7(b). This code is transformed by adding an outer *while* loop and the unlikely *if* block is replaced by a *break* statement. When the condition *count* > *times* is True, the outer *for* loop will break and the *if* statement in new while loop is entered. After execution of this block, the *for* loop continues running from the iteration it left off. The outer *for* is now DOALL.

5 Results

We implemented the algorithms discussed in the previous section in the OpenIMPACT compiler [24]. The algorithms identify opportunities for control dependence speculation (DOALL-uncounted loops), register reduction variable expansion, long distance memory dependence iteration chunking, speculative loop fission, speculative prematerialization, and infrequent dependence isolation. For reduction variable expansion, the compiler identifies reduction variables for summation, production, logical AND/OR/XOR, as well as variables for finding min/max. For speculative loop fission, the compiler identifies loops where the sequential part represents less than 3% of the dynamic execution of the loop. For infrequent dependence isolation, the frequency of cross iteration register dependence and library calls has to be less than 10%. For long distance memory dependences, a threshold of 4 iterations is assumed. These thresholds were experimentally chosen to maximize the speedup. Several benchmarks from SPEC CPU 92/95/2000, MediaBench [16], and Unix utilities are studied and we show the results for all the benchmarks that successfully ran through our system.

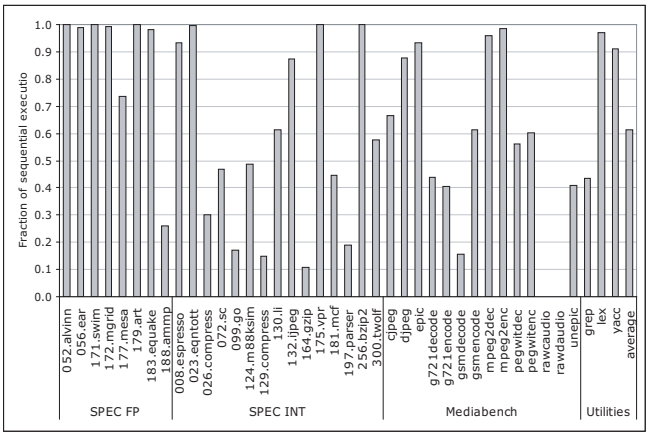

Figure 8: Fraction of sequential execution covered by speculative DOALL loops after transformations

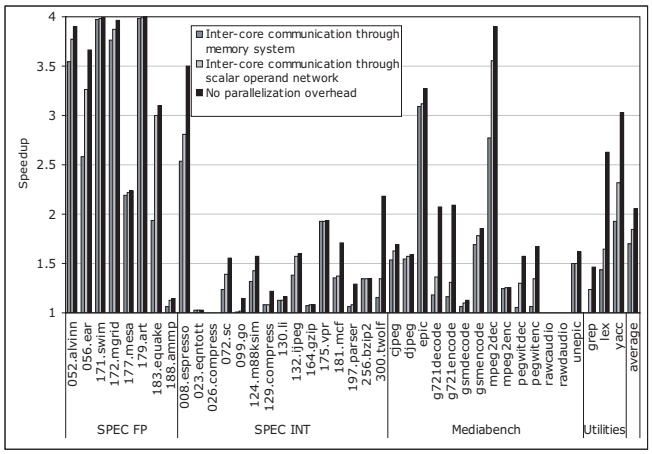

Figure 10: The first two bars show the effect of inter-core communication through memory systems versus scalar operand network. The last bar shows the speedup without any parallelization overhead.

We use a multicore simulator that models 1-8 single-issue in-order processor cores similar to the ARM-9. The simulator models a scalar operand network, where we assume a 3 cycle latency to communicate register values between neighboring cores. A perfect memory system is assumed. We use a software transactional memory [8] to emulate the underlying hardware transactional memory described in Section 3. We assume that the transaction abort incurs an average of 500 cycles overhead and requires execution of the reset block as well as re-execution of the chunk. Successfully committed transactions do not incur extra overhead besides XBEGIN and XCOMMIT instructions.

Figure 8 shows the fraction of the dynamic sequential execution that can be parallelized after applying all our proposed transformation techniques. On average, 61% of the sequential execution can be speculatively parallelized. This number is more than twice the coverage of 28% using TLS without any transformation, and 8.5 times the 7% gain by relying on static analysis alone (see Figure 1). More importantly, for SPECint benchmarks, the transformations are able to uncover loop level parallelism in 55% of the sequential execution, where previous techniques yield poor results.

Figure 9 shows the speedups achieved on 2-core, 4-core and 8-core machines compared to a single core. One stacked bar is shown for each configuration. The lower part shows the speedup results without our proposed transformations. Therefore, only counted loops with no or very few cross iteration dependences from profiling are parallelized. The higher part of the stack bar shows the speedup after all transformations are applied. The compiler chooses the most profitable loop to parallelize if multiple nesting levels can be parallelized.

On average, we achieved a speedup of 1.36 with transformations compared to the 1.19 speedup without transformation on a 2-core machine. In addition, we got much higher speedups on 4-core and 8-core configurations as the average speedup increases from 1.41 to 1.84 for the 4-core machine and from 1.63 to 2.34 for the 8-core machine after applying transformations.

The speedup values vary widely across different benchmarks. For SPECfp benchmarks, significant speedup values are achieved due to the inherent parallel nature of the applications. In the SPECint benchmarks, the average speedups are 1.19, 1.37 and 1.50 for 2-core, 4-core and 8-core configurations, which is a considerable improvement over previous techniques. As shown in the figure, the baseline compiler cannot extract much parallelism from these benchmarks due to the large number of inter-iteration register and control dependences typically found in C applications.

The speedups for MediaBench benchmarks are generally higher than SPECint, while the Unix utility benchmarks have similar results to SPECint. If we consider reduction variable expansion as part of the baseline, the speedups without transformations increase to an average of 1.24, 1.59 and 1.95, respectively. Our new transformations still achieve considerable speedups on top of that. It should also be noted that reduction variable expansion helped mostly in the SPECfp benchmarks compared to the integer applications. The new transformations are especially helpful for SPECint benchmarks where traditional transformations alone are largely unsuccessful. Note that high coverages shown in Figure 8 do not always translate to high speedup numbers. Mpeg2enc is an example of such a case. The loops identified as parallel in this example have a small loop body and low trip count. Therefore, the parallelization overhead makes it much less appealing for parallelizing such loops. Another observation is that the SPECfp benchmarks are quite scalable. Several benchmarks such as 171.swim, 172.mgrid and 179.art achieve almost linear speedup on 4 and 8 cores.

As mentioned before, we use Scalar Operand Network (SON) to communicate register values between the cores and maintain commit orderings. In Figure 10, we studied the effects of SON and also the code generation framework on the benchmark speedups. This figure shows the speedup values on a 4-core machine with different configurations. Three bars are shown for each benchmark. The first bar shows the speedup assuming we don't have a scalar operand network and all communications between the cores must go through the shared L2 cache. In this scheme, the commit order is maintained using locking primitives. We assume communication between cores takes 30 cycles. The second bar shows the case when we have a SON can communicate register values between neighboring cores with a 3-cycle latency. This is the same data from Figure 9. As shown in the figure, for some benchmarks such as 171.swim, 179.art and 175.vpr, we can achieve nearly the same speedup with and without the SON. These benchmarks have large loop bodies with high trip counts, and the communication overhead is amortized by large chunks of parallel work. On the other hand,

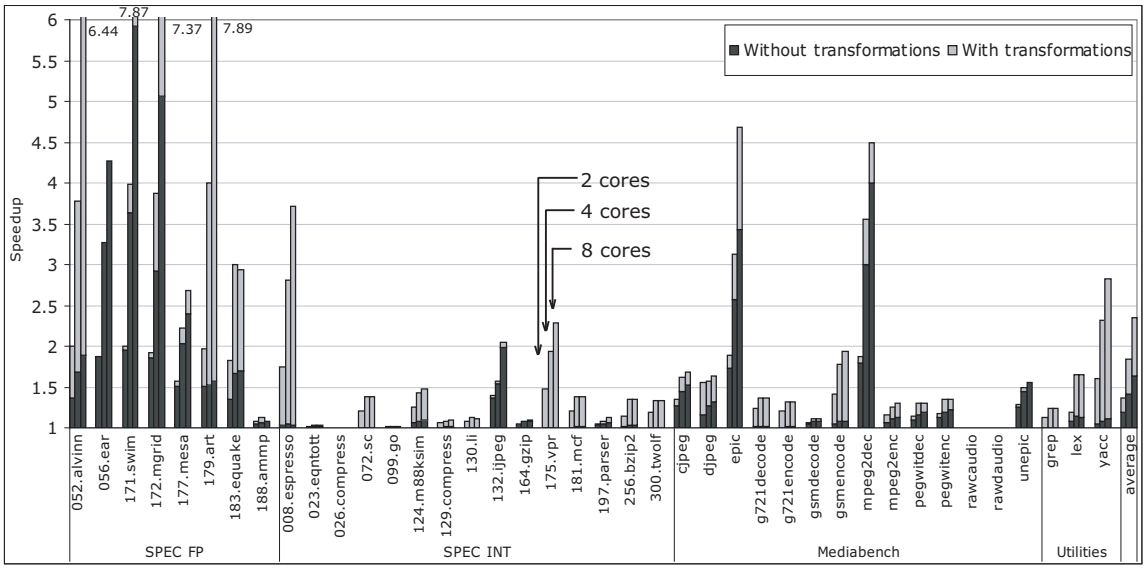

Figure 9: Effect of the compiler transformations on performance for 2, 4, and 8 core systems. The bottom portion of each bar shows the baseline with TLS only and the top portion shows the combined affects of all transformations.

benchmarks, such as pegwitdec, pegwitenc, and grep, suffer significantly from lack of the SON. The loop sizes in these benchmarks are small to medium, and the high communication overhead easily takes away all the benefits resulting from parallelization. On average, the speedup without the SON is 1.70 compared to the 1.85x speedup with the SON. We can see that a reasonable benefit can be achieved from the transformations without a SON as well. The third bar in Figure 10 shows speedups without the parallelization overhead. Our speculative parallelization framework decreases the performance gain by about 10% on average compared to the configuration with zero parallelization overhead.

Table 1 shows the the fraction of the dynamic sequential execution that can be parallelized with different techniques. It also shows the abort frequencies during speculative execution. The first 7 columns show the coverage of different transformation techniques. Each element represents the time spent in loops that are parallelized by a certain transformation. For example, by applying reduction variable expansion in 052.alvinn, loops that account for 97% of the sequential execution time can become parallelizable. For the transformation in each column, we assume the techniques in columns to the left have been applied, so the data in a column represents the additional parallelism being uncovered. If both the outer and inner loop in a nested loop are parallelized by a certain technique, only the time spent in the outer loop taken into account. If the outer and inner loop are parallelized by two different techniques, the time spent in the inner loop will show up in two categories. Therefore, the numbers in a row could add up to more than 100%.

The first column shows the coverage of parallel loops without any transformations. The second column shows the fraction of time spent in DOALL-uncounted loops. On average, control flow speculation alone enables an additional 9% of the sequential execution to execute in parallel. Moreover, it is an enabling technique for other transformations. For example, infrequent dependence isolation converts a loop into a nested loop, and the inner one is DOALL-uncounted. The third column is for register reduction variable expansion. On average, it converts 15% of the execution into parallel loops. This is a relatively simple transformation and provides good performance improvement. For any system that tries to parallelize sequential programs, reduction variable expansion should be the first low hanging fruit to go for.

The next column shows fraction of time spent in loops that can be parallelized by speculative loop fission. On average, it enables 28% of the execution to partially execute in parallel. This transformation has the largest potential among the ones that we studied. It is especially useful for SPECint benchmarks for which other techniques do not provide much benefit.

Prematerialization, infrequent dependence isolation, and iteration chunking for long distance memory dependence each improve parallel coverage significantly for certain benchmarks. In contrast to control speculation, reduction expansion and fission, they each affect less than 50% of the benchmarks. However, in the benchmarks which they show benefits, it is usually quite significant. On average, they improve the parallel coverage by 4%, 5% and 8%, respectively.

The last column in the table shows the abort frequency during speculative execution. The abort frequency is the number of aborted iterations divided by the total number of iterations in the benchmark. As shown in the figure, they are quite low, and most benchmarks have abort frequencies less than 2%. We also studied the stability of the profile results on different inputs. We found loops without cross iteration memory dependence on one input usually do not have cross iteration memory dependence using other inputs as well. As a result, speculative DOALL loops are mostly consistent across different inputs.

6 Related Work

There is a large amount of previous work in TLS [2, 13, 14, 25, 30, 32, 34, 39] and TLDS [31, 32] that propose speculative execution of threads along with the required architectural support. For example, Multiscalar architectures [30] support TLS, and prior work [36] has studied graph partition-

Bench	DOALL	CS	RE	SF	PM	IDI	LD	AF
SpecFP								
052.alvinn	55	0	97	94	0	0	0	0
056.ear	99	0	0	0	0	0	0	0
171.swim	98	0	2	98	0	0	0	0
172.mgrid	98	0	0	97	0	0	0	0
177.mesa	68	0	6	0	0	0	0	0
179.art	51	0	77	100	94	77	0	0
183.equake	66	0	41	16	0	44	0	0
188.ammp	10	16	0	1	0	0	0	0
SpecINT								
008.espresso	8	12	18	9	25	3	22	2
023.eqntott	2	95	0	30	0	1	97	0
026.compress	1	0	0	29	0	0	0	0
072.sc	1	3	14	5	0	24	0	0
099.go	4	2	0	8	0	3	1	1
124.m88ksim	15	1	0	34	0	34	0	0
129.compress	2	0	9	4	0	0	0	0
130.li	0	0	0	17	0	0	44	1
132.ijpeg	60	19	1	16	9	26	0	0
164.gzip	9	0	0	2	0	0	0	1
175.vpr	0	0	0	79	0	0	100	0
181.mcf	1	1	0	42	0	0	1	2
197.parser	12	0	2	6	0	0	1	1
255.vortex	0	0	0	0	0	0	0	0
256.bzip2	5	1	0	100	0	0	49	2
300.twolf	1	0	0	54	0	2	1	0
MediaBench								
cjpeg	53	0	2	9	0	6	0	0
djpeg	29	23	23	59	23	0	0	1
epic	86	0	87	85	0	0	1	0
g721decode	8	0	36	0	0	0	0	0
g721encode	7	0	34	0	0	0	0	0
gsmdecode	12	3	0	0	0	0	0	0
gsmencode	12	2	47	0	0	1	0	0
mpeg2dec	92	5	67	5	0	3	0	0
mpeg2enc	13	84	33	31	10	0	0	0
pegwitdec	26	0	0	31	0	0	0	0
pegwitenc	28	0	0	32	0	0	0	0
rawcaudio	0	0	0	0	0	0	0	0
rawdaudio	0	0	0	0	0	0	0	0
unepic	41	0	10	12	0	0	0	0
Utilities								
grep	0	43	0	0	0	0	0	0
lex	21	5	0	70	0	2	0	0
yacc	33	52	17	2	0	3	20	6
average	27	9	15	28	4	5	8	0

Table 1: Percentage sequential code coverage of various transformations – Last column shows the Abort frequencies in the benchmarks. Coverages higher than 20% are highlighted. (CS:control speculation for uncounted loop, RE: reduction expansion, SF: speculative fission, PM: prematerialization, IDI: infrequent dependence isolation, LD: ignore long distance dependence, AF: abort frequency).

ing algorithms to extract multiple threads; however, this does not eliminate unnecessary dependences in the same way this work does. Our work builds upon previous research and proposes compiler transformations to expose more speculative parallelism in loops. In particular, the Hydra project [25] classifies loops to different categories and introduces compiler techniques to parallelize the code. This work extends those ideas with compiler techniques for loop identification, selection, transformation, and code generation. MSSP [39] transforms code into master and slave threads to expose speculative parallelism. It creates a master thread that executes an approximate version of the program containing a frequently executed path, and slave threads that run to check results. Conversely, our transformations have different execution models. Both speculative fission and infrequent path isolation create parallel threads executing different iterations. No dedicated checker threads are needed.

Several works have proposed full compiler systems [2, 9, 18, 27, 31] that target loop-level and method-level parallelism. In [18], the authors introduce a compilation framework for transformation of the program code to a TLS compatible version. Profile information is also used to improve speculation choices. The Mitosis compiler [27] proposes a general framework to extract speculative threads as well as pre-computation slices (p-slices) that allow speculative threads to start earlier. Our prematerialization is similar to p-slices, but prematerialization is highly targeted to loop recurrences that can be unwound to decouple iterations and must maintain register and control dependences, while p-slices can speculatively prune paths. Du et al. [9] propose a compilation framework in which candidate loops for speculation are chosen based on a profile-guided misspeculation cost. A general compilation strategy for TLS is introduced in [2]. Their method is applicable to loops as well as other parts of the program. Due to the application of this general approach, many opportunities in loop transformation and parallelization are skipped. Chen et al. [6] use pointer analysis to figure out memory dependences. However, this sophisticated pointer analysis prevents full characterization of memory accesses in the program. Also, as mentioned before, our work transforms many loops to make them more parallelizable. We extend previous work in that we studied a comprehensive set of existing and new transformations to expose more parallel opportunities hidden under removable dependences.

The LRPD Test [29] and variants [21] speculatively parallelize DOALL loops that access arrays and perform runtime detection of memory dependences. These techniques work well for codes that access arrays with known bounds, but not general single-threaded programs.

Speculative decoupled software pipelining (DSWP) [35] presents another technique for thread extraction on loops with pointer-chasing cross-iteration dependences. DSWP pipelines a single iteration across multiple cores. This has the advantage of overlapping the sequential and parallel portions. However, when the two portions are not relatively equal sized, the pipeline can be unbalanced. Our approach has benefits in load balancing and scalability, particularly for small recurrence cycles. Further, DSWP checkpoints architectural state for every iteration in flight using a versioned memory. Storage grows with the length of the pipeline. A separate commit thread synchronizes the memory versions and handles commits. Conversely, speculative fission uses a conventional transactional memory where only one buffer per core is required and no commit thread.

The JPRM [5] framework uses a dynamic approach for loop parallelization. Although this might lead to more accurate speculation, the overhead of dynamic binary manipulation might become too high. Furthermore, dividing the loop too chunks with a length of one iteration incurs a significant bookkeeping overhead.

Previous work also researched exploiting fine-grain parallelism in loops. Lee et. al. [17] studied running loop iterations simultaneously with communications through register channels. This technique is good for loops with cross-iteration dependences that cannot be removed through transformations. Our previous work [38] studied exploiting fine-grain parallelism in a single iteration, which is orthogonal to this work and can be applied simultaneously.

7 Conclusion

The microprocessor industry is gradually shifting from single-core to multicore systems. A major challenge going forward is to get the software to utilize the vast amount of computing power and adapt single-threaded applications to keep up with this advancement. In this paper, we studied the automatic parallelization of loops in general-purpose applications with TLS hardware support. By studying the benchmarks, we found that a considerable amount of loop-level parallelism is buried beneath the surface. We adapted and introduced several code transformations to expose the hidden parallelism in single-threaded applications. More specifically, we introduced speculative loop fission, isolation of infrequent dependences, and speculative prematerialization to untangle cross iteration data dependences, and a general code generation framework to handle both counted and uncounted speculative DOALL loops. With our transformation techniques, more than 61% of dynamic execution time in the general applications can be parallelized compared to 27% achieved using traditional techniques. On a 4-core machine, our transformations achieved 1.84x speedup compared to 1.41x speedup without transformations.

8 Acknowledgments

We thank the anonymous referees for their valuable comments and suggestions. We also thank Neil Vachharajani, Easwaran Raman, Arun Raman from the Liberty research group for their help with the simulation system. This research was supported by the National Science Foundation grants CNS-0615261 CCF-0347411, and the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: 14th Annual Symp. on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[3] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[4] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. 7th PACT*, page 176, Oct. 1998.

[5] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. 30th ISCA*, pages 434–446, 2003.

[6] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proc. 8th PPoPP*, pages 25–36, June 2003.

[7] K. Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, Feb. 1993.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. 2006 International Symposium on Distributed Computing*, 2006.

[9] Z.-H. Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. '04 PLDI*, pages 71–81, 2004.

[10] R. Gupta. Employing register channels for the exploitation of instruction level parallelism. In *Second PPoPP*, pages 118–127, 1990.

[11] R. Gupta. A fine-grained MIMD architecture based upon register channels. In *Proc. 23rd Annual Workshop on Microprogramming and Microarchitecture*, pages 28–37, 1990.

[12] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[13] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *8th ASPLOS*, pages 58–69, Oct. 1998.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proc. '04 PLDI*, pages 59–70, June 2004.

[15] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, NY, 1978.

[16] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th MICRO*, pages 330–335, 1997.

[17] S. Lee and R. Gupta. Executing loops on a fine-grained MIMD architecture. In *Proc. 24th MICRO*, pages 199–205, 1991.

[18] W. Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. 11th PPoPP*, pages 158–167, Apr. 2006.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. 25th MICRO*, pages 45–54, Dec. 1992.

[20] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proc. 8th HPCA*, page 55, Feb. 2002.

[21] S. Moon, B. So, and M. W. Hall. Evaluating automatic parallelization in SUIF. *JPDC*, 11(1):36–49, 2000.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th HPCA*, pages 254–265, Feb. 2006.

[23] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. 11th SAS*, pages 165–180, Aug. 2004.

[24] OpenIMPACT. The OpenIMPACT IA-64 compiler, 2005. http://gelato.uiuc.edu/.

[25] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, Feb. 1997.

[26] M. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proc. 10th PPoPP*, pages 142–152, June 2005.

[27] C. G. Quinones et al. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. '05 PLDI*, pages 269–279, June 2005.

[28] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proc. 13th PACT*, pages 177–188, 2004.

[29] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Trans. on Parallel and Distributed Systems*, 10(2):160, 1999.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. 22nd ISCA*, pages 414–425, June 1995.

[31] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *Trans. on Computer Systems*, 23(3):253–300, 2005.

[32] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. 4th HPCA*, pages 2–13, 1998.

[33] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. 31st ISCA*, pages 2–13, June 2004.

[34] J. Tsai et al. The superthreaded processor architecture. *IEEE Trans. Comput.*, 48(9):881–902, Sept. 1999.

[35] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *Proc. 16th PACT*, pages 49–59, Sept. 2007.

[36] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proc. 31st MICRO*, pages 81–92, Dec. 1998.

[37] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. 13th HPCA*, pages 261–272, Feb. 2007.

[38] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proc. 13th HPCA*, pages 25–36, Feb. 2007.

[39] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. 35th MICRO*, pages 85–96, Nov. 2002.