Extracting Statistical Loop-Level Parallelism using Hardware-Assisted Recovery

Steven A. Lieberman, Hongtao Zhong, and Scott A. Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor
{lieberm,hongtaoz,mahlke}@umich.edu

Abstract

Chip multiprocessors with multiple simpler cores are gaining popularity because they have the potential to drive future performance gains without exacerbating the problems of power dissipation and hardware complexity. These designs provide real benefits for server-class applications that are explicitly multi-threaded. However, for desktop and other systems, there is a large code base of single-thread applications that have yet to see any benefit from multicore systems. While these applications were designed for execution on a single processor, many regions of computation have statistically suitable structure for extracting thread-level parallelism. In this work, we examine automatic extraction of statistical loop-level parallelism from single-thread applications. Our approach is to utilize simple hardware mechanisms combined with intelligent compiler code generation to perform low-cost targeted thread-level speculation. Our technique combines memory dependence profiling to identify suitable loops, stylized code generation to untangle register dependences between iterations, and a hybrid compiler/hardware recovery mechanism to handle infrequent memory dependences. We show that significant amounts of statistical loop-level parallelism indeed exist in non-numeric applications, and present the architectural extensions and detailed compiler algorithms required to exploit it.

1 Introduction

For more than four decades, the semiconductor industry has depended on Moore's law to deliver consistent application performance gains through the multiplicative effects of increased transistor counts and higher clock frequencies. However, power dissipation and thermal issues have emerged as dominant design constraints and caused architects to shy away from relying on increasing clock frequency to improve performance. Exponential growth in transistor counts still remains intact and a powerful tool to improve performance. Semiconductor companies now put two to eight cores on a chip, and this number is expected to continue growing.

One of the key challenges going forward is software: if the number of devices per chip continues to grow consistently 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 [15, 6, 3, 9, 14]. 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. Explicit parallel programming is one potential solution to the problem. However, these systems may burden the programmer with implementation details and can severely restrict productivity and creativity. 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 [27, 22, 10, 29, 31, 2, 33, 13]. 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, 18, 7, 17]. Traditional TLS techniques have two important weaknesses. First, TLS architectures introduce a high amount of hardware complexity and area overhead. The job of detecting dependence violations, rolling back processor and memory state, and initiating re-execution is all placed on the hardware, which in the general case requires substantial complexity. Second, the overhead of spawning and executing threads is high, thus its crucial to identify large chunks of independent work [7, 17]. Conversely, general-purpose applications often contain frequent branches and loops with small bodies and low average trip counts.

For this paper, we take a different direction to TLS. A compiler/hardware technique is proposed that combines low-cost hardware that is explicitly managed in software and a stylized compiler code generation strategy for targeting specific, high-rate-of-return speculation opportunities. We partition the responsibilities between hardware and software. Hardware is responsible detecting memory dependence violations and rollback of the memory state. Conversely, software is responsible for managing all register dependences, register rollback, and spawning threads. Recovery from mis-

speculation is jointly handled. By using a combined approach, the hardware cost and complexity of TLS can be substantially reduced. Further, providing hardware structures that are explicitly managed by the software enables more efficient thread spawning and recovery. In general purpose applications where thread sizes are often modest, efficient thread management is key to achieving performance gains.

Our technique targets automatic extraction of loop-level parallelism (LLP). Loops with sets of completely independent loop iterations, or DOALL loops, are identified and parallelized. However, sophisticated memory dependence analysis, such as points-to analysis [20], is generally ineffective for uncovering DOALL loops in general-purpose applications. Thus, this work identifies loops that have statistically independent iterations for parallelization. Memory dependence profiling is used to gather statistics on memory dependence patterns in candidate loops. Loops that are highly unlikely to have cross-iteration dependences are selected and split across multiple cores. In this manner, the misspeculation rate and recovery penalties are kept extremely low.

One of the main challenges with exploiting statistical DOALL parallelism across multiple cores is the compiler code generation. Profiling can identify loops with unlikely memory dependences, but often loops in general-purpose applications have complex register dependence patterns that must be untangled in order to effectively parallelize iterations. In particular, register live-ins and live-outs are privatized for each core. Further, induction and accumulator variables must be replicated and separated for each group of loop iterations that are executed together. Traditional techniques, such as scalar expansion where scalar variables are mapped to separate memory variables, are not effective due to the overhead of accessing memory. Rather, all privatization and expansion optimizations are performed at the register level. Loops in general-purpose applications also tend to have small bodies. Loop iterations are distributed to the cores in chunks to reduce thread spawning overhead and make better use of the data cache.

In the event of an unexpected memory dependence violation, the recovery responsibilities are split between the compiler and hardware. A lightweight transactional memory system detects dependence violations and provides rollback capabilities of the memory state for each core. Register state is rolled back by having compiler create a self-initializing block on each core to reset all relevant register values to their initial state and restart loop execution. By shifting a significant portion of the TLS responsibilities to the software, unnecessary hardware complexity can be eliminated and a more efficient solution achieved.

2 Opportunities for Statistical LLP

2.1 Predictability of Memory Dependences

Although scientific applications often contain loops whose memory access patterns can be identified and proven by the compiler, existing compilers have difficulty proving the absence of cross-iteration memory dependences in general purpose applications due to their extensive use of pointers, recursive data structures, and complex control flow. However, parallelization opportunities still exist in these applications; a significant fraction of loops show zero or few cross-iteration memory dependences during execution.

To determine the viability of predicting which loops are parallelizable based on profiling information, we performed a study of the nature of cross-iteration memory dependences. Register dependences are not considered because they are obvious to the compiler. For all loops in a program, we profile to identify cross-iteration memory dependences. (Details on the profiler can be found in Section 4.1). If two memory operations from different iterations access the same address, and one is a store, there is a cross-iteration dependence. For each loop, we can then obtain the fraction of iterations with a cross-iteration dependence, we call this the *dependence fraction*. Figure 1 shows histograms of how much time is spent in loops with different dependence fractions across various applications in the SPEC and MediaBench suites. The left set of bars for each benchmark (gray bars) show the distribution of dependence fractions for a training input. The y-axis represents how much serial execution time was spent in loops with a particular dependence fraction; the percentage is out of the total execution time spent in loops. The portion of the benchmark spent within loops is shown at the top of each graph in parenthesis after the benchmark name.

Examining this structure, we hypothesize that the loops with zero dependences are unlikely to have dependences for other input sets, and would be good candidates for parallelization. To check this, we profiled only these loops using an evaluation input for the benchmark; the result is also presented in Figure 1. The right set of bars (dark gray bars) show the distribution of dependence fractions of the loops that we expect to be zero (since they were zero in the training input). For most benchmarks, the hypothesis is strongly supported. One notable exception is 256.bzip2, where one loop representing 7% of the execution changes its memory dependence behavior significantly with the larger input set.

One particular concern in systems that hardware to detect such memory dependences (e.g., coherent caches) is false sharing. The data in Figure 1 was consistent across cache line sizes of 4 to 64 bytes. While this is not a comprehensive study, we conclude that false sharing does not significantly affect the identification of statistical DOALL loops. This confirms the false sharing results presented in [29]. With highly predictable memory dependences, these loops are promising candidates for speculative parallelization; however, we have thus far ignored the feasibility of handling register dependences.

2.2 Opportunities for Statistical DOALL Loops

We now consider a class of loops where loop-level parallelism can be exploited by the compiler and the rate of speculation recovery is expected to be low. A *statistical DOALL loop* is one where all register dependences can be removed via compiler transformations, and the memory profile shows that there are zero or very few cross-iteration memory dependences.

Figure 2 illustrates two statistical DOALL loops taken from SPEC applications. Figure 2(a) shows a loop from the benchmark 256.bzip2. The compiler cannot prove this loop is DOALL because the value of the variable `a2update` cannot be determined at compile time. Hence, the statement `quadrant[a2update] = qVal` could access the same memory location in different iterations. However, an execution profile shows that iterations access different memory locations at runtime, thus the loop is statistically likely a DOALL. The loop can be parallelized to speed up the execution if there exist mechanisms to detect potential cross-iteration memory dependence violations and recover when

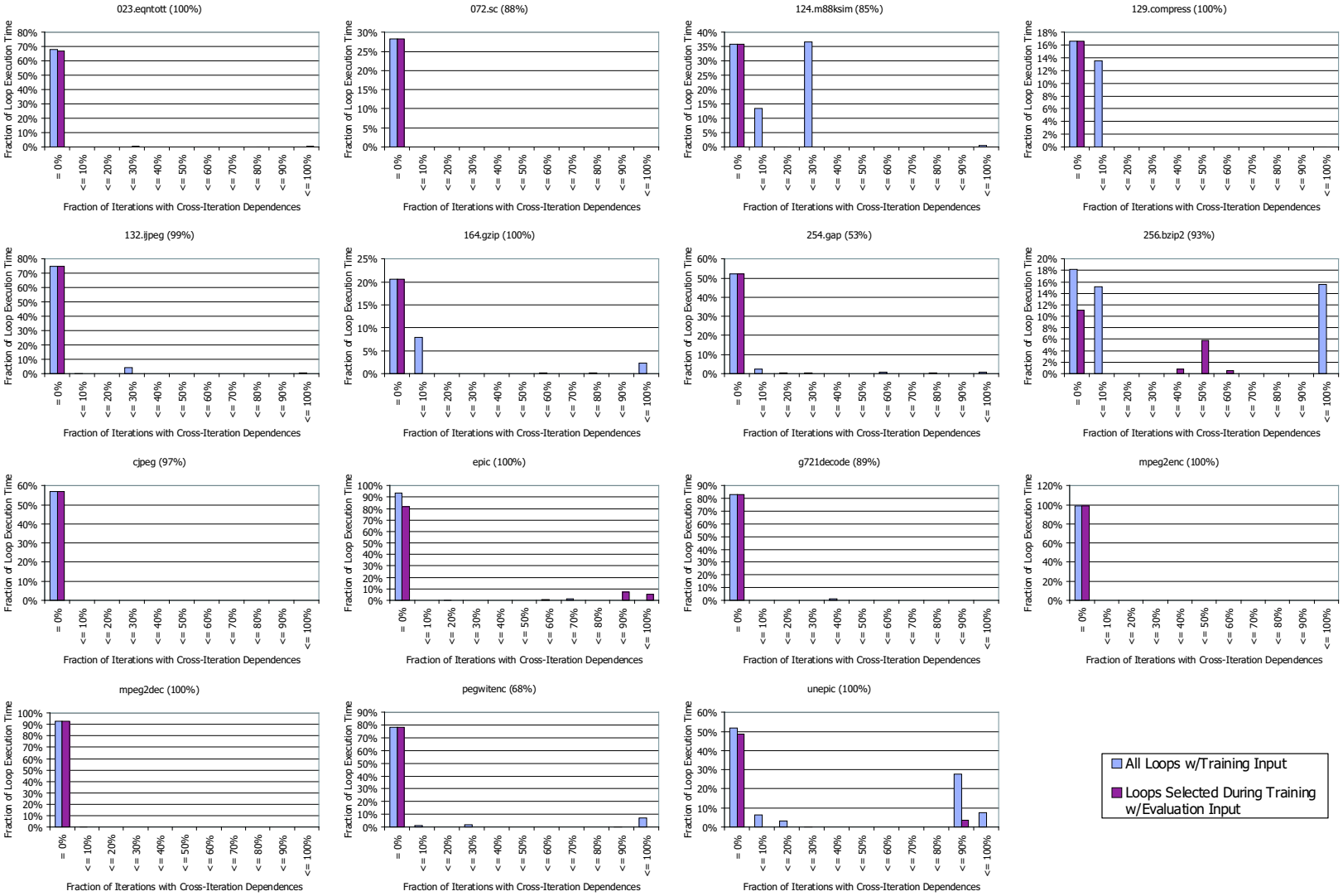

Figure 1: Distribution of execution time for loops with varying fractions of iterations with cross-iteration dependences. The left bar represents the breakdown of all loops for a training input. The right bar represents the breakdown of the zero cross iteration dependence loops on the training input using a larger, evaluation input for the benchmark.

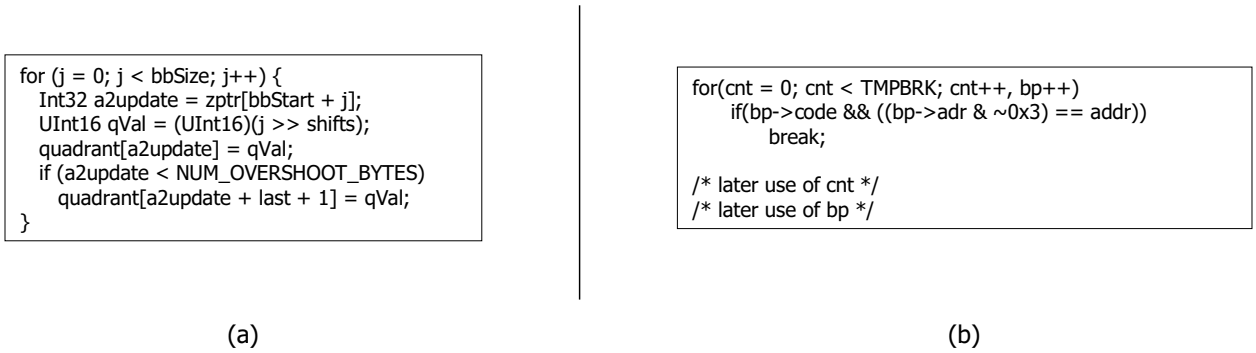

Figure 2: Statistical DOALL loops: (a) Loop from 256.bzip2 is a DOALL-counted due to an unlikely memory dependence, (b) Loop from 124.m88ksim is a DOALL-uncounted due to a cross-iteration control dependence.

a violation is detected. Because the number of iterations is known when the execution enters the loop, this type of loop is referred to as *DOALL-counted*.

Figure 2(b) shows a loop from 124.m88ksim. This loop is not DOALL because of a cross-iteration control dependence. The execution of later iterations depends on whether any prior iteration executed the break statement. Aside from the control dependence, the loop does not have any cross-iteration dependences. If the loop executes many iterations before exit, it is worth being parallelized. Since the number of iterations is unknown when the execution enters the loop, this type of loop is referred to as *DOALL-uncounted*. DOALL-uncounted loops require additional mechanisms to handle discarding the side effects of unnecessary iterations.

To estimate the potential benefit of parallelizing statistical DOALL loops, several SPEC and MediaBench applications were manually examined with the help of profile information. Figure 3 shows the fraction of serial runtime spent in parallelizable loops (both provable and statistical) for the benchmarks. The fraction of provable DOALL loops are estimated using the memory dependence analysis tool in the OpenIMPACT compiler [21]. This figure shows the fraction of execution time covered by three types of parallelizable loops: provable DOALL, statistical DOALL-counted, and statistical DOALL-uncounted.

As shown in the figure, the opportunity to parallelize using DOALL loops varies across the benchmarks. Some benchmarks, such as 023.eqntott, epic and mpeg2enc, have more than 80% of their execution in DOALL loops, while 072.sc, 129.compress, 164.gzip, 254.gap and 256.bzip2 spend a more modest fraction of time (less than 20%) in DOALL loops. The time spent on different types of DOALL loops also varies across benchmarks. For example, mpeg2dec has more than 30% of its execution time in provable DOALL loops. Provable DOALL loops are generally restricted to loops that exclusively access arrays with affine indices. Given the extensive use of pointers, provable DOALL loops are not common in non-numeric applications. Even MediaBench applications often use pointers and dynamic memory allocation, thereby making the loops difficult to provably parallelize. However, a significant fraction of loops are statistically DOALL. Epic has spends more than 80% of its execution time in statistical DOALL-counted loops, and more than 90% of the execution time for 023.eqntott is spent in DOALL-uncounted loops.

Overall, the data shows that there are large number of parallelization opportunities using statistical DOALL loops for many benchmarks. The rest of this paper studies low-cost techniques to exploit these statistical parallelization opportunities.

3 Architectural Support for Statistical LLP

This paper proposes mechanisms to exploit statistical loop level parallelism with low hardware cost. The spirit is to expose several generic architectural features to the compiler and allow the compiler to manage speculative thread spawning and recovery. Hardware is used when the software alternative is too costly. For example, hardware is used to detect dynamic memory dependence conflicts. A number of works [29, 27, 10] have proposed executing speculative threads in environments where much of the burden is given to hardware; these require complex hardware to fully support TLS. By employing a hardware/software approach, our technique has a lower cost and overhead compared to hardware-centric techniques. Admittedly, hardware-centric TLS techniques can support speculative execution of both loops and acyclic code, while we only considered using our approach for loops in this work.

3.1 Requirements for Statistical DOALL Execution

Efficiently exploiting statistical loop-level parallelism requires several mechanisms to facilitate speculative execution, which can be implemented in hardware, software, or a combination. System designers must decide how much of the burden should be given to hardware, and how much to software and the compiler. The following is a list of features required to efficiently execute statistical DOALL loops.

Requirement 1: Detection of cross-core memory dependences. If two memory operations access the same location, and one is a store, a dependence exists between those operations.[1] Loops are parallelized if the profile indicates that cross-iteration memory dependences are unlikely. At runtime, accesses to the same memory location from different cores must be detected and a recovery action initiated.

Requirement 2: Undo/rollback. If a memory dependence violation is detected, the core executing the higher

[1]Note that in more complex systems, system designers may opt to eliminate anti and output dependences via hardware, thus they would be invisible to software and not treated as dependences.

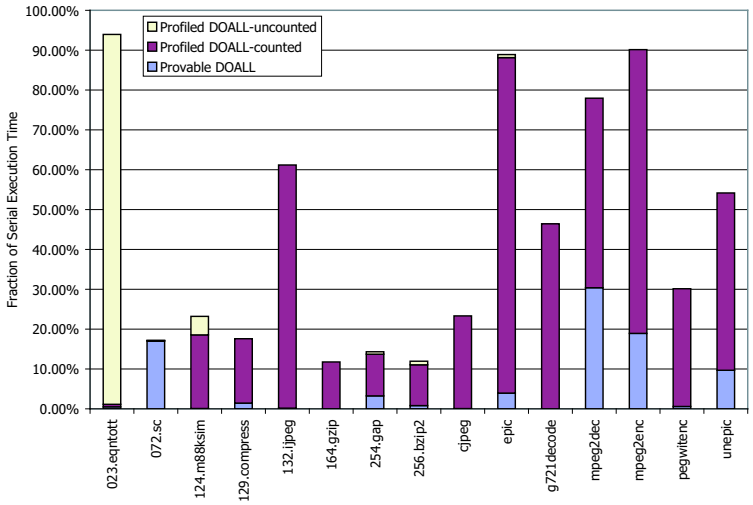

Figure 3: Loop-level parallelism available in SPEC INT and MediaBench [16] applications.

numbered iterations needs to abort and restart. The architectural state on the aborted core, including registers and memory contents, must be rolled back to the state before the speculative execution started.

Requirement 3: Lightweight thread spawning. To exploit statistical loop level parallelism, multiple threads, each consisting of one or more loop iterations, execute speculatively in parallel on multiple cores. A low latency mechanism to spawn threads is essential for the performance of statistical DOALL loops, especially those with small bodies and low trip counts.

Requirement 4: Inter-core scalar communication. Before the parallel execution of a loop, scalar register values are passed to all cores to initialize live-in values. Similarly, after the parallel execution, live-out scalar values of the loop are passed from all cores to the users of the live-out scalars. A low latency mechanism to pass scalar values between cores is essential, especially for loops with low trip count.

Requirement 5: Instruction ordering between cores. During the speculative execution of loop iterations, certain actions must occur in order between cores. For example, cores must commit their speculative results in the original program order. This ordering is required for every loop, so an efficient way to enforce ordering among cores is needed.

3.2 Target Architecture

Figure 4(a) shows the overall structure of the proposed architecture for statistical DOALL execution. The architecture is a standard chip multiprocessor with two extensions. First, hardware transactional memory [12, 24, 11] is added to detect memory dependence violations and roll back memory state. The rollback of registers is handled by the compiler. Second, a scalar operand network, similar to that in RAW [30], is added to allow direct communication of scalars, support fast thread spawning, and enforce instruction ordering between cores.

The system has four in-order cores. Each core has private L1 instruction and coherent data caches. All four cores share an L2 cache and main memory. The cores access a unified memory space; the coherence of caches is handled by a bus-based snooping protocol. A scalar operand network connects the cores in a 2-D mesh. Details about the transactional memory and the scalar operand network are presented next.

Low-Cost Transactional Memory. The inter-core memory dependence detection and rollback of memory state are supported by a transactional memory. A transaction is a segment of code running on one core, marked by the programmer or the compiler, that appears to execute atomically when observed by other cores. Transactional memory supports parallel execution of transactions on multiple cores. The memory system monitors addresses accessed by each core (taking advantage of cache coherence), and aborts and restarts one or more transactions if the transactional semantics are violated. Transactional memory is a good fit for statistical DOALL execution because the iterations assigned to each core can be viewed as a transaction; as long as the transactions are committed in the correct order, the loop appears to execute sequentially.

The low-cost transactional memory implementation used here is inspired by early work on transactional memory [12]. There is a large body of research on using transactional memory to support TLS; Garzarán et al. [8] provide a thorough survey of techniques. This implementation falls on the lowest-cost end of their spectrum (only one copy of each memory location exists system-wide, and each core only executes one speculative transaction at a time). Three new instructions are needed in the ISA, as shown in the top portion of Table 1. Each cache block's state is augmented with a speculative bit. A transaction begins with an XBEGIN instruction, which specifies an abort handler. Once the transaction begins, the speculative bit will be set on any cache block accessed (via a load or store). Note, previously dirty blocks are flushed back to memory or to a write buffer upon XBEGIN. During a transaction, all loads and stores must appear atomically to other processors. We detect the following situations via the coherence network, if one of these situations occurs, at least one of the cores involved must be aborted:

- Any core that loads a value with its speculative bit set in another core's cache and was dirty
- Any core that stores a value with its speculative bit set

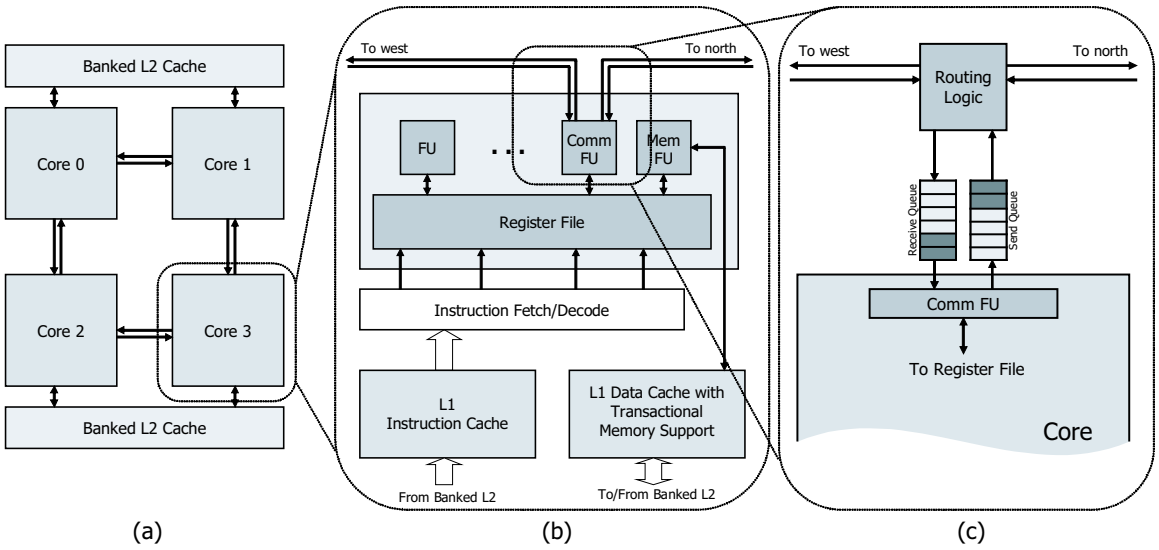

Figure 4: Block diagram of the target architecture: (a) 4-core system connected in a mesh topology, (b) Datapath of an individual core, (c) Details of the inter-core communication function unit.

in another core's cache

- Speculative cache capacity is exceeded: any cache block is evicted from the data cache and any victim caches

To decide which core should abort, we simply abort the core with the higher core ID, since the compiler parallelizes a loop using a template where higher iterations are always executed on a higher number core. Note, however, that systems such as those that resolve such conflicts with timestamps [24] or transaction IDs, would also suffice. The aborted processor will invalidate all blocks marked speculative and jump to its abort handler. The compiler inserts code to roll back register state in the abort handler. This will be discussed in Section 4.4.1.

One particular concern is cache capacity overflow. As described later (Section 4.4.1), the compiler can try to avoid capacity overflow; however, there must be a mechanism to ensure forward progress in case of overflow. In case of an abort due to overflow, our architecture aborts all transactions and serializes them. Other techniques that support arbitrarily large transactions, such Unbounded Transactional Memory [1] and Virtualizing Transactional Memory [25], can also be used to ensure forward progress.

Unlike other transactional memory implementations, the transactional memory used here supports an ABORT instruction that allows a core to abort transactions on other cores. It takes a core ID and an abort_handler as source operands. Any pending transactions on the specified core will be aborted and the control on that core will jump to the address specified by the abort_handler. This is needed to parallelize DOALL-uncounted loops; in these loops, if one iteration exits the loop, it needs to stop the execution on all higher cores.

Our compiler technique allows us to use a transactional memory in one of its simplest forms. Much of the work on TLS [29, 17] assumes the memory system will resolve anti and output dependences; however, this requires the memory to keep track of the bytes, instead of the cache lines, that are modified, which incurs significant hardware cost. Profile data in Section 2 has shown that there plenty of statistical loop level parallelism can be exploited without support for removing anti and output dependences. Thus, hardware savings can be achieved by not designing hardware to catch the infrequent dependence case, while still allowing exploitation of loops with unlikely memory dependences.

Scalar Operand Network. Inter-core communication is supported by a scalar operand network, similar to that in the RAW processor [30]. The network allows cores to communicate scalar values between register files. Each core has a communication function unit (Comm FU) that allows it to access the network. As shown in Figure 4(c), the Comm FU contains send and receive queues and simple routing logic (XY routing is assumed). Two new instructions, SEND and RECV, are added to the instruction set, as seen in the bottom portion of Table 1. 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. The scalar network allows low latency communication of register values between cores.

SEND and RECV operations also provide a mechanism to guarantee the ordering of instructions in different cores. For example, if instruction A in core 0 should be executed before instruction B in core 1, the compiler can insert a SEND after A in core 0 and a RECV before B in core 1. The SEND and RECV communicate a dummy value. Since the RECV stalls the core if the corresponding data has not arrived, B is guaranteed to execute after A.

The scalar operand network also provides a natural mechanism for lightweight thread spawning in a master/slave configuration. The master core executes the main program. When it wants to spawn a thread on a slave core, it simply sends a program counter (PC) value to that core. The slave core then starts its execution from the received PC.

(a)

Instruction	Behavior
XBEGIN *abort_handler*	Starts a transaction, specifying a program counter of an abort handler. All loads and stores will now mark accessed cache blocks as speculative. If a conflict occurs, marked lines will be invalidated and control will jump to the abort handler.
XCOMMIT	Commits a transaction. This clears the speculative markings from all cache lines (thus making their values non-speculative).
ABORT *core_id, abort_handler*	Causes a transaction abort on the specified core, control on the specified core will jump to the abort_handler.

(b)

SEND *core_id, src_reg*	Takes the value from *src_reg* and sends it via the network to *core_id*. Stalls if the send queue is full.
dest_reg = RECV *core_id*	Retrieves the first queued value from *core_id* and writes it to *dest_reg*. Stalls if the value is not available.

Table 1: Instructions needed to (a) control low-cost-transactional memory, and (b) utilize the scalar operand network.

Since the compiler explicitly controls thread spawning, the live-in scalar values for the slave are explicitly passed from the master using SEND/RECV instructions. When the slave thread completes, live-out scalar values are explicitly communicated to the master thread, and the slave will enter a sleep mode and wait for the next start PC. The master and slave threads share the same stack frame, so the transactional memory guarantees that the accesses to the shared stack frame from multiple cores will not conflict, otherwise the slave thread will be aborted. This spawning mechanism is lightweight because it does not require any stack manipulation, and software only copies the required register state.

Hardware/Software Breakdown of Requirements. The proposed architecture, together with appropriate compiler support, meets all five requirements listed in Section 3.1. Table 2 shows how the burden of meeting those requirements is divided between hardware and compiler. A low-cost transactional memory is used to detect memory dependence violations and rollback memory state when necessary. For register state, the hardware transfers control to the address specified by abort_handler when a memory dependence violation is detected, the compiler is responsible for implementing the abort_handler and recovering the necessary register values for re-execution. A scalar operand network in the hardware is used to support fast thread spawning, low latency inter-core scalar communication, and enforce instruction ordering. The compiler must insert code to send PC and live-in scalar values to spawn a thread, insert SEND/RECV instructions to communication scalar values between cores, and insert SEND/RECV instructions that communicate dummy values to enforce ordering of instructions between cores. Details of the compiler code generation are presented in the next section.

4 Compilation for Statistical LLP

The goal of our compiler is to identify statistical DOALL loops, select the loops that are profitable to parallelize, and generate a parallel version of the code for selected loops to achieve speedup. Our compiler can parallelize two types of loops: statistical DOALL-counted and DOALL-uncounted. Figure 5 shows the work flow of our compiler. It first performs profiling on the program to collect memory dependence information for all loops. Then, the loop classifier identifies loops as not DOALL, *DOALL-counted*, and *DOALL-uncounted*. A heuristic is used to select loops that are profitable to parallelize. Finally, the selected loops are transformed and assembly code is generated to parallelize the loops. The following sections describe each of these steps.

4.1 Profiling

The compiler profiles the program to collect cross-iteration memory dependence information. We profile memory accesses to find loops with no or very few cross iteration memory dependences as candidates for parallelization. The memory profiler emulates the program with a training input set. Since a single instruction may be within multiple nested loops, the profiler is cognizant of which iterations the execution is currently in for all loop nests. For each recently accessed memory address, the profiler stores the last instruction to access it, and the loop iterations in all nest levels at the time of last access. (We found that storing only the last 512k recently accessed addresses was sufficient and kept profiling runs generally under 10 minutes.) This allows the profiler to identify if a dependence occurred across iterations. The compiler also collects the execution frequency of every basic block in the iteration profile to facilitate heuristic loop selection and other parts of the the compiler.

4.2 Loop Classification

The loop classifier classifies the loops into three categories: *DOALL-counted*, *DOALL-uncounted*, and not DOALL. If a loop has no cross-iteration dependences, or only contains removable cross-iteration dependences, it is DOALL. We further categorize loops as either DOALL-counted loops, where the number of iterations can be determined upon loop entry, or as DOALL-uncounted loops, which can have any loop termination condition. DOALL-counted and DOALL-uncounted loops can be parallelized using different compiler techniques.

The classifier first examines the profile for cross-iteration memory dependences. If such dependences exist, the loop is not DOALL. Theoretically, if the profile shows infrequent cross-iteration memory dependences, the loop can still be classified as statistical DOALL; this would require the loop

Requirement	Hardware Support	Compiler Support
1. Memory Dependence Detection	Transactional memory.	N/A
2. Rollback	For memory: transactional memory. For register: jump to abort handler.	For memory: N/A For register: implement the abort handler to roll back registers.
3. Lightweight Spawning	Scalar operand network.	Insert code to send PC and live-in scalar values.
4. Scalar Communication	Scalar operand network.	Insert SEND/RECV to communicate scalar values.
5. Instruction Ordering	Scalar operand network.	Insert dummy SEND/RECV to guarantee instruction ordering.

Table 2: Breakdown of hardware/software responsibilities for efficient execution of statistical DOALL loops.

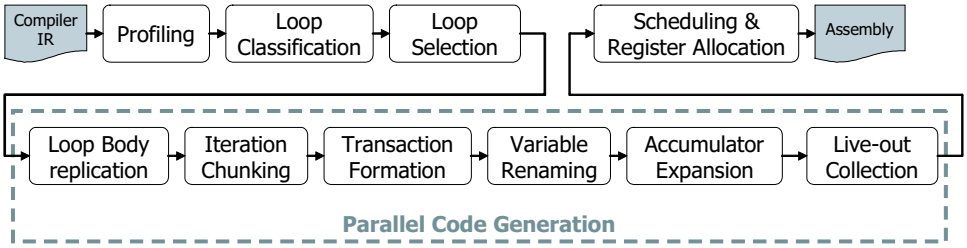

Figure 5: Parallelization flow

selection heuristic to accurately estimate the cost of these rare dependence violations. For this paper, we focus on the loops show no cross-iteration dependences during profiling, and leave loops with infrequent memory dependences for future work.

Next, the classifier examines all cross-iteration register dependences and determines if they can be eliminated via compiler transformations. All output dependences and anti-dependences can be eliminated, as well as some flow (true) dependences, such as accumulator operations and induction variables.

The cross-iteration register dependences that can be removed are:

- **Local variables** are variables that are always defined before being used in the loop body. In the loop shown in Figure 2(a), both `a2update` and `qVal` are local variables. Cross-iteration anti-dependences exist for local variables. These dependences can be removed by renaming the local variable for each core.
- **Write-only variables** are defined in the loop body but not used in the loop body. They are live-out of the loop, so there are cross-iteration output dependences for write-only variables. These dependences can be removed by renaming the variables for each core, and recording whether the variables were written on that core, so that the last written value can be used upon loop exit.
- **Induction variables** are variables that are updated by the same increment on every loop iteration, such as the variable `j` in Figure 2(a). The value of an induction variable in any iteration can be calculated from its initial value, the increment, and the iteration number.
- **Reduction variables**, such as accumulators or variables used to find the maximum or minimum value, causes cross-iteration flow (true) dependences. An example of accumulator variable is in `sum = sum + a[i]`, where `sum` is an accumulator variable. These dependences can be removed by creating a local accumulator (or min/max variable) for each core, and accumulating the local accumulators (or finding the global min/max among local min/max) after the loop exit.

If a loop has no other cross-iteration dependences, the loop is *DOALL*. The test for counted loops is then applied. For a loop to be counted, the number of iterations to be executed in the loop must be known when the loop execution starts. It requires that all conditional branches that exit the loop are (a) based on the induction variable, (b) have the same comparison condition, and (c) compare it to the same loop-invariant value. Loops that pass this test are *DOALL-counted*, and all others are *DOALL-uncounted*.

4.3 Heuristic Loop Selection

Simply parallelizing every DOALL loop can sometimes reduce overall application performance because of the parallelization overhead. This overhead includes the extra instructions to remove register dependences, chunk loop iterations, supply live-in values, and fix up live-out values. If the parallelization overhead outweighs the potential time savings, the loop should not be parallelized.

A heuristic is employed that estimates the overall speedup possible by parallelizing the loop. First, serial runtime is estimated using profile information. Then, the parallelization overhead required to divide the work amongst multiple cores is estimated. This is done by evaluating the loop to see

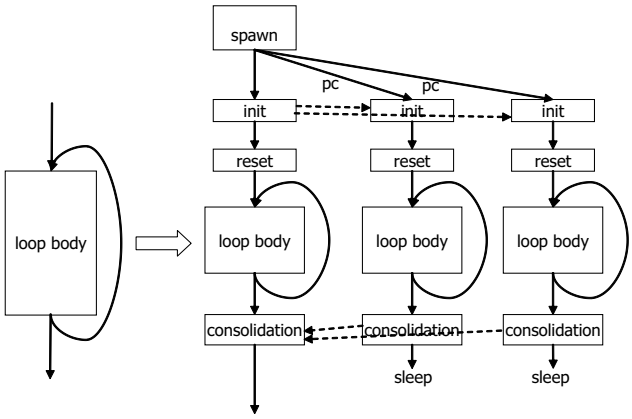

Figure 6: Parallelizing a loop for a 3-core system. The loop body is replicated, and initialization blocks, reset blocks, and consolidation blocks are added as detailed in Section 4.4.1. Dashed lines represent inter-core data communication.

what cross-iteration dependences need to be removed, and estimating the additional instructions required to handle each dependence. Costs are specified in terms of number of operations, but send/receive pairs are counted as three operations to estimate communication overhead. These parallelization operations are executed once per loop invocation, and the total is called the $parallelization\_overhead$. Third, we determine c, the average number of parallel chunks of work. If the average number of iterations is higher than the number of cores, c is simply the number of cores; otherwise, it is the average number of iterations. The parallel runtime (excluding overhead) is optimistically estimated as the serial runtime divided by c. Finally, from the serial runtime, parallel runtime, and $parallelization\_overhead$, we compute the estimated speedup. The loop is parallelized if the estimated speedup is higher than a threshold; in this work, 1.15 is used since our estimate is optimistic.

4.4 Code Generation

Once loops have been identified as good candidates for parallelization, the compiler must divide the loops' work across multiple cores. For all loops, the compiler must replicate the loop body for each core, insert setup and cleanup code, and remove register dependences. First, Section 4.4.1 discusses code generation for DOALL-counted loops, and then Section 4.4.2 shows how the compiler techniques can be extended to handle DOALL-uncounted loops.

4.4.1 Code generation for DOALL-counted loops

To parallelize DOALL-counted loops, the compiler performs the six parallel code generation steps shown in Figure 5. The first three steps create the framework for parallelization, and the next three untangle additional dependences within this framework.

The loop body is first replicated. The original copy of the loop is deemed the copy for core0, and an additional copy of all basic blocks in the loop is made for each additional core. Figure 6 shows how the loop body is replicated across the cores. (The additional blocks shown in the figure have not been inserted yet.)

Second, iteration chunking is performed. For cache locality, it is usually desirable to place contiguous iterations on the same core. To divide the iterations between cores, induction variables are privatized for each core. New operations are inserted in a new preheader block, the *initialization block* (as seen in Figure 6), to compute the initial, final, and increment values for each core. For simplicity, this computation is done on core0 and communicated to other cores.[2] Secondary induction variables, which have initial but no final values, are treated similarly; their initial values are computed based on those of the primary.

The third step is transaction formation: the compiler must explicitly manage speculative register state, as well as ensure that speculative memory state is committed correctly. The compiler forms a framework to allow undo/rollback of register states and to commit transactions in order. Two new basic blocks are created for each core: a *reset block* and a *consolidation block*. A reset block is created before the loop (as shown in Figure 6) to allow transactions to restart; register values can be reset without re-executing the inter-core communication in the initialization block. The reset block begins the transaction with an XBEGIN instruction and specifies its own PC as the abort handler. Then, instructions in the reset block duplicate any live-in values that will be modified in the loop body into new registers, so that their original values can be restored should the transaction restart. As dependences are untangled in later stages, any operations that should be re-executed if the transaction restarts are inserted into the restart block.

Figure 7 provides a more detailed picture using a loop from 256.bzip: iteration chunking instructions have been inserted in the initialization blocks, and XBEGIN and MOVE instructions have been inserted in the reset blocks.

A consolidation block is inserted after the loop; it is responsible for both committing the transactions and consolidating any register values live after loop termination. Transactions must be committed in order, so SEND and RECV instructions are inserted, as illustrated in Figure 7. The consolidation block on the first core contains an XCOMMIT followed by a SEND of some dummy value to the second core. The second core contains a corresponding RECV, followed by an XCOMMIT, and finally a SEND to the third core. This scheme enforces the order of the iterations because the RECV will stall the core if the SEND hasn't been executed. Conceptually, chunks of iterations are being committed as atomic updates.

The fourth step in parallel code generation is variable renaming. Local variables can be renamed, removing anti-dependences. Values that are live-in, but never modified, can be replicated; a local copy is made for each core, and communicated from core0 in the initialization block.

Fifth, accumulator operations are expanded to local accumulators. Local accumulators are created for each core and initialized in the *reset block* (so they are reset if an abort occurs). The local accumulator is communicated back to core0 in the consolidation block (after transaction commit), and core0 performs the final reduction operation.

Finally, output dependeces for live-out variables are untangled. Since live-out values may be conditionally updated, the compiler creates a predicate on each core for each live-out register. The predicate is initialized to False in the reset block, and is set to True when the register has been set on that core. In the example of Figure 7, assume that `qVal` is

[2]This typically takes approximately 14 arithmetic operations for each core, but when loop count information is statically available, global classic optimizations in later phases of the compiler eliminate these operations.

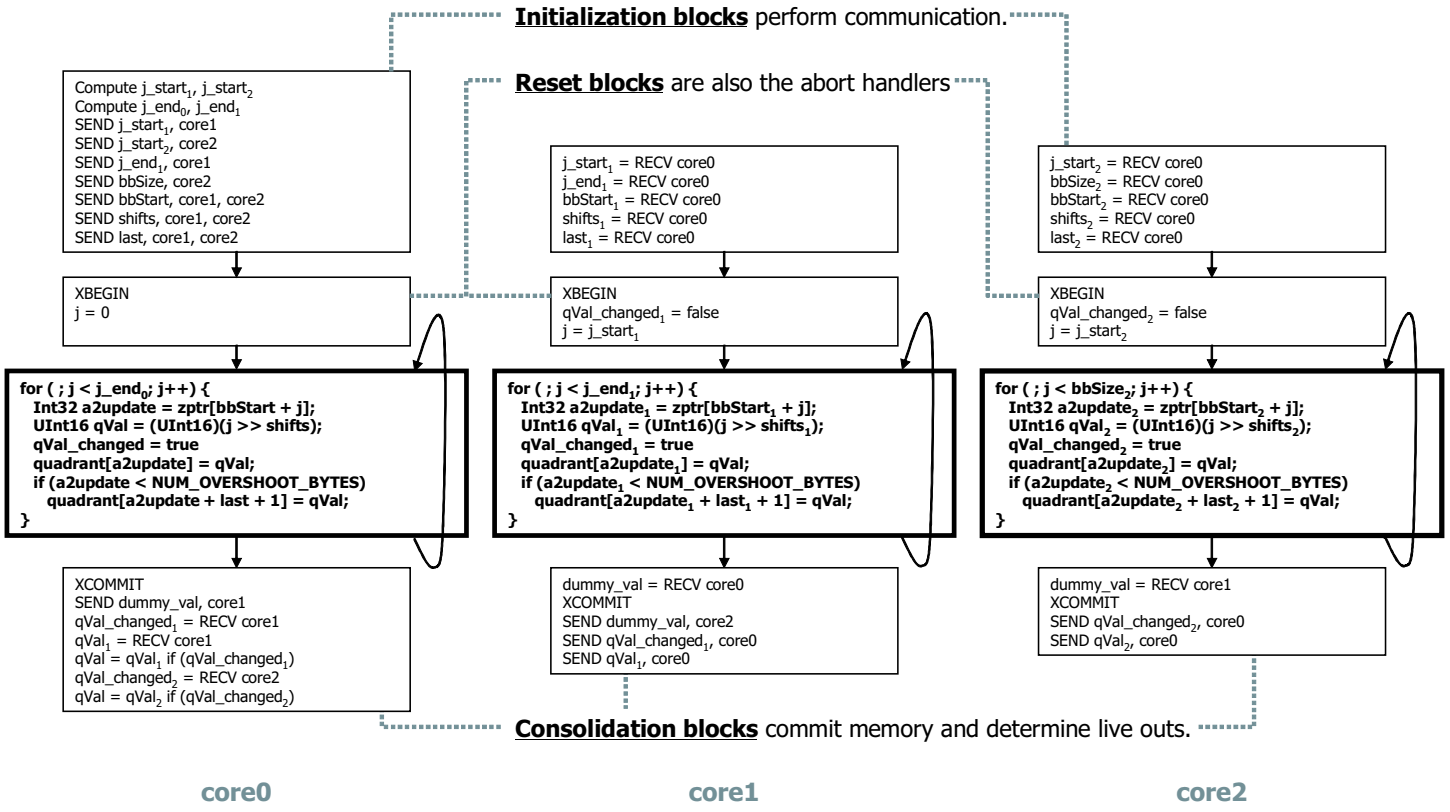

Figure 7: Parallelized code for a DOALL-counted loop in 256.bzip2 for three cores.

live-out for illustration purposes. The compiler then create a new predicate `qVal_changed`. Operations are added to set the predicate to True in all basic blocks that update the live-out register. Then, each live-out register can be renamed to a local live-out. The predicates and local live-out values are communicated back to core0 in the consolidation block. The block on core0 has a sequence of MOVE operations that move the received values back into the original live-out register based on the received predicates, as detailed in the figure. Live out variables are generally the most expensive dependence to untangle, but the heuristic step already accounts for this.

This completes the parallel code generation. One remaining concern is the underlying architectural capacity for speculative state. Transactional memories have a limited capacity. Our architecture guarantees forward progress by aborting all transactions and serializes them. Overflow can also be prevented by dividing the loop into smaller chunks if a loop is determined to access a large number of memory addresses via profile information. An outer loop can be created to iterate through the sets of smaller chunks.

4.4.2 Code generation for DOALL-uncounted loops

Additional transformations are necessary for DOALL-uncounted loops, where the exact number of iterations is not known. Our approach divides the iteration space into fixed size chunks. Chunks can be executed in parallel across cores. An outer while-loop is created to iterate through the chunks. If an iteration meets the exit condition, the execution in all higher cores are aborted, and a predicate is set to terminate the outer while-loop.

Figure 8(a) shows a DOALL-uncounted loop in in 300.twolf, and Figure 8(c) provides a high-level picture of the transformation for this loop. The inner loop in the figure contains num_cores fixed size chunks. The profile information is used to decide an appropriate chunk size for the loop based on the average number of iterations. The inner loop is treated as a DOALL-counted loop. Local variables, live-in values, live-out values, and accumulators are handled in a similar manner to DOALL-counted loops. The communication of live-in values can be done outside the newly-created outer loop. Live out values and accumulators are communicated within the outer loop (but outside the inner loop); this allows updating the live out register state when transactions are committed. If any iteration in the inner loop takes the early exit, all execution in higher cores should be aborted. The compiler inserts ABORT instructions before loop early exits to stop execution on higher cores.

An outer loop is created to iterate the chunks. The compiler creates a new predicate, called $exited$, as the exit condition for the outer loop. The predicate is initialized to False outside the outer loop. The predicate $exited$ becomes True if any iteration in the inner loop takes the early exit. A local predicate $exited_i$ is created for each core i, An instruction is inserted before each loop early exit branch that sets $exited_i$ to True whenever the branch would be taken. $exited$ is computed as a logical OR of all $exited_i$s.

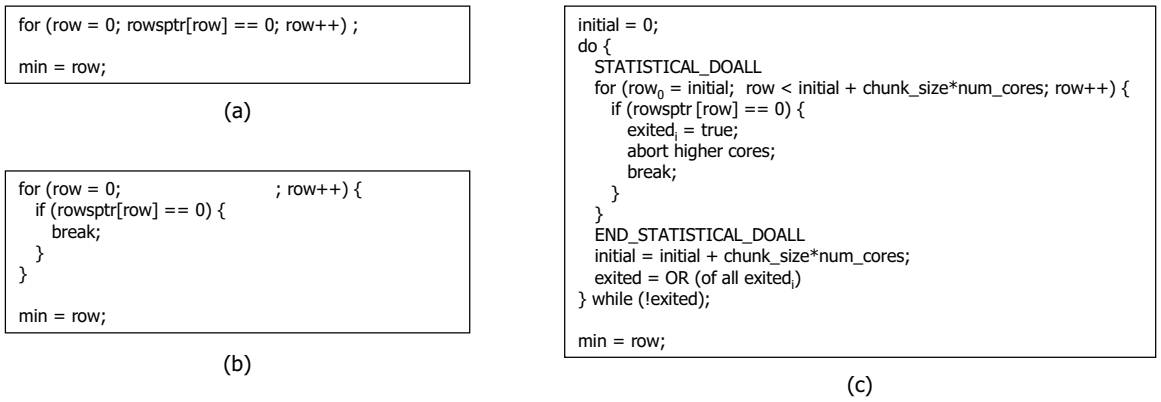

Figure 8: (a) A statistical DOALL-uncounted loop in 300.twolf (b) An equivalent version of the same loop that more closely resembles matches assembly code. (c) The loop structured for parallelization as a DOALL-uncounted.

5 Related Work

The LRPD Test [26] and variants [19] speculatively paralellize 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-thread programs, and they show speedup on specific loops in scientific codes.

Previous work on Thread-Level Speculation (TLS) [22] and Thread-Level Data Speculation (TLDS) [29, 28] proposes the execution of threads with architectural support. Those works propose programmer identification of regions of code that can form transactions, and also discuss speculation on loops. Oplinger et al. [22] in particular identify different styles of loops. This work extends those ideas with compiler techniques for loop identification, selection, and automatic code generation.

Several works have also proposed full compiler systems [17, 7, 2] that target loop-level parallelism (and sometimes method-level parallelism). However, there are two key differences between these compilers and our own. First, these compilers rely on a more complex memory system to untangle anti and output dependences, whereas our work will attempt to handle such dependences in software, if they can be promoted to registers. Second, our work uses profiling to test all loop nests simultaneously, and replicates code only when necessary to achieve iteration chunking, rather than relying on ILP-targeted unrolling. The JPRM [5] compiler framework does profile all loop nests simultaneously, but does so using hardware, and it also relies on a more complex memory system.

Multiscalar architectures [27] also support thread-level speculation, and prior work [32] has studied graph partitioning algorithms to extract multiple threads; however, this does not eliminate unnecessary dependences in the same way this work does.

Decoupled software pipelining (DSWP) [23] presents another technique for (non-speculative) thread extraction on loops with pointer-chasing cross-iteration dependences. The loops amenable to DSWP are disjoint from the loops studied here, so their techniques are orthogonal to this work.

6 Conclusion

This paper has shown that there is significant potential for statistical loop-level parallelism in many single thread applications. Compiler techniques were presented that show general code generation strategies for these important classes of loops that are applicable to many architectures with memory speculation support.

It is important to remember that these techniques only focus on a certain class of loops. Even though in our system we executed all non-loop code on a single core, there is still plenty of potential for parallelism in that code. By combining an arsenal of techniques for different styles of code regions, we believe automatic extraction of speculative threads has a promising future.

References

[1] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: 14th Annual ACM Symposium 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. of the 7th International Conference on Parallel Architectures and Compilation Techniques*, page 176, Oct. 1998.

[5] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.

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

[7] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[8] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Vinals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2(3):247–279, 2005.

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

[10] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, page 102, June 2004.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[13] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.

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

[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. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[17] W. Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, Apr. 2006.

[18] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, page 55, Feb. 2002.

[19] S. Moon, B. So, and M. W. Hall. Evaluating automatic parallelization in SUIF. *Journal of Parallel and Distributed Computing*, 11(1):36–49, 2000.

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

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

[22] 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.

[23] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, 2005.

[24] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[25] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, June 2005.

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

[27] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

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

[29] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[30] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 341–353, Feb. 2003.

[31] J. Tsai et al. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, Sept. 1999.

[32] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 1998.

[33] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.