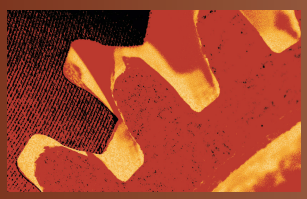
Speculative Parallelization

Arturo González-Escribano and Diego R. Llanos
University of Valladolid, Spain



Some emerging technologies try to exploit the parallel capabilities of modern processors.

Parallel systems are increasingly common. Thanks to the use of chip multiprocessor (CMP) architectures, even commodity desktop processors offer multicore technologies.

These capabilities make the system faster by executing several processes at once. However, because compilers are still largely unable to extract the parallelism inherent to sequential code, it's difficult to take advantage of parallel hardware with a sequential application.

Numerous parallel languages, parallel extensions to sequential languages, and library functions are available for developing parallel code. However, manual parallelization of a sequential algorithm is still a difficult and error-prone task.

Taking advantage of a parallel system's capabilities requires a deep understanding of the algorithm itself, the programming model being used, and the underlying architecture for specific optimizations.

The lack of a universally accepted parallel model for programs or machines and the huge amount of sequential code developed so far make the idea of automatic parallelization extremely attractive.

COMPILER-BASED AUTOMATIC PARALLELIZATION

To execute an algorithm in parallel, the first problem to solve is determining which parts of the code the compiler can parallelize.

Most parallelizing compilers focus on loop-level parallelism, studying how to execute different iterations simultaneously. To be parallelized, two iterations shouldn't present any data dependency—that is, neither should rely on calculations that the other one performs. As Figure 1 shows, if no dependences are found, the compiler can safely parallelize the loop.

Unfortunately, in many cases, the compiler can't determine whether two sets of iterations are independent. For example, the code in Figure 2 presents a dependency between two iterations. The particular iterations involved depend on the value of a variable k, whose value is unknown at compile time.

In this situation, the compiler conservatively refuses to execute the loop in parallel, although in our three-threads example, some values of k such as 3, 6, or 9 don't produce cross-thread dependences. Sequential code that uses complex subscript or pointer arithmetic leads to similar situations.

Despite these difficulties, compiler-based automatic parallelization is a major trend for parallel programming. The sequential programmers' expertise, accumulated across decades, and the huge legacy of sequential code are important reasons for supporting this approach.

SPECULATIVE PARALLELIZATION

The most promising technique for automatically parallelizing loops when the system cannot determine dependences at compile time is speculative parallelization. Also called thread-level speculation, this technique assumes optimistically that the system can execute all iterations of a given loop in parallel.

A hardware or software monitor divides the iterations into blocks and assigns them to different threads, one per processor, with no prior dependence analysis. If the system discovers a dependence violation at runtime, it stops the incorrectly computed work and restarts it with correct values. Of course, the more parallel the loop, the more benefits this technique delivers.

To better understand how speculative parallelization works, it is necessary to distinguish between private and shared variables. Informally speaking, private variables are those that the program always modifies in each iteration before using them. On the other hand, values stored in shared variables are used in different iterations.

If all variables are private, no dependences can arise, and the system can execute the loop in parallel. Shared variables can lead to dependence violations only if a value is written in a given iteration and a successor has already consumed an outdated value.

Figure 2 shows this situation, which is known as read-after-write dependence. To handle this dependence, speculative parallelization discards and reexecutes the consumer thread and all threads that execute subsequent blocks with the new, correct values. Figure 3 shows this process, which is called a *squash operation*. Note that some subsequent threads may indeed be correct, but the over-

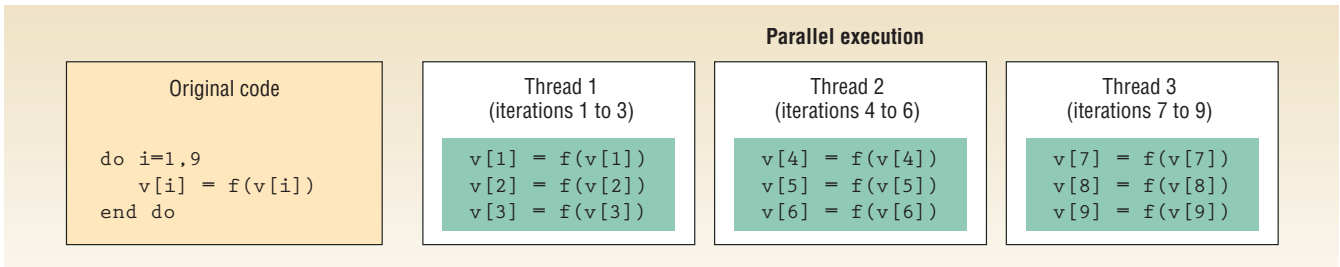


Figure 1. Parallel execution of a loop with no dependences using three threads. Because all iterations of the loop are independent, the compiler can issue them in parallel.

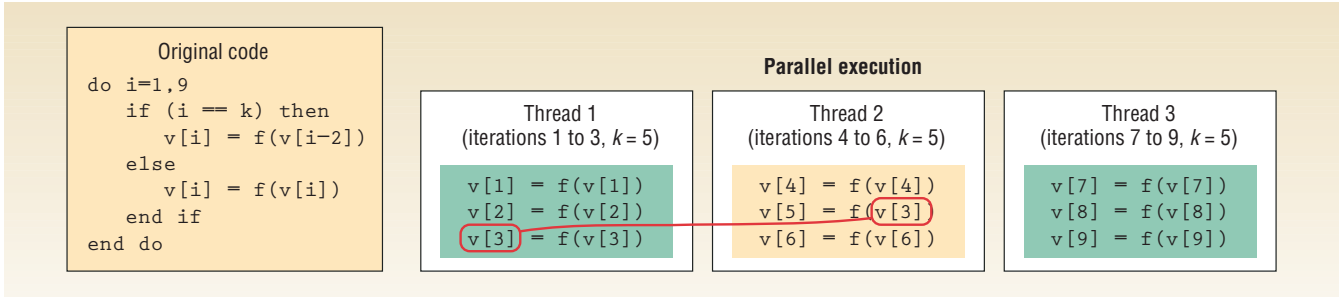


Figure 2. In this loop, a value k *not known at compile time prevents loop parallelization by the compiler: If* k = *5, thread 1 may not have computed the value of* v*[3] in time for use by thread 2. Note that if the dependence did not cross thread boundaries (for example, with* k = *6), the compiler could parallelize the loop.*

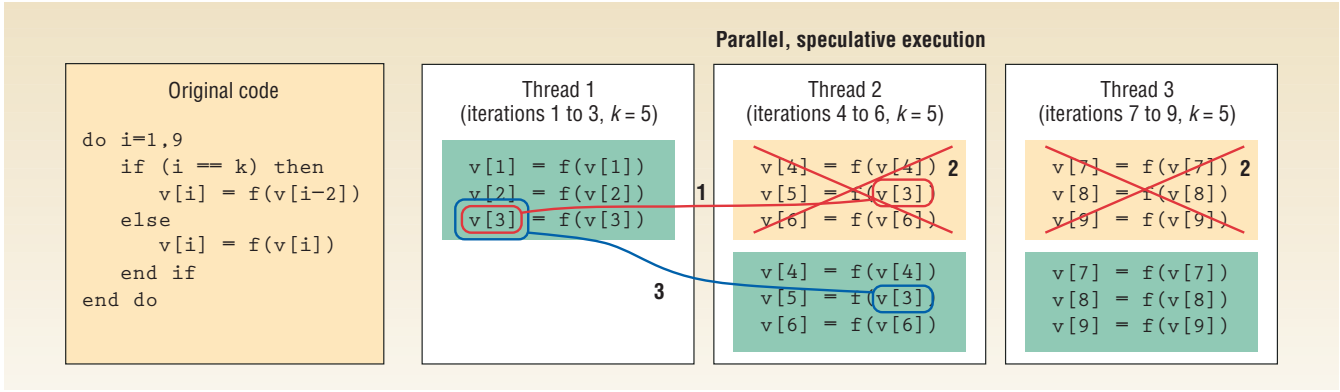


Figure 3. Speculative parallelization starts the execution in parallel of the loop, while a monitor tracks the execution to detect cross-thread dependence violations. If such a violation occurs, (1) speculative parallelization stops the consumer thread and all threads that execute subsequent blocks, (2) discards its partial results, and (3) restarts the threads to consume the correct values.

head of checking which ones are correct is usually too high.

To simplify squashes, threads that execute each block of iterations don't directly change the shared variables; instead, each thread maintains its own version of those variables. Changes are reflected to the original shared variables through a commit operation only if the execution of all iterations up to this point succeeds. However, if the thread is squashed, the system discards its version of the data.

Either a hardware or software monitor can detect dependence violations and issue all the auxiliary operations, such as spawning and stopping threads. Hardware solutions rely on additional hardware modules to detect dependences, while software methods augment the original loop with new instructions that check for violations during the parallel execution. When a dependence violation occurs, the monitor performs the squash operation and restarts the execution of the consumer threads.

It's easy to see that speculative parallelization's effectiveness depends on how many iterations the system can execute in parallel. If the loop is mostly sequential, the overhead that the monitor expends during the speculative execution can result in performance losses with respect to the sequential version of the same algorithm. Even if the loop is fully parallel, the overhead can make the speculative parallelization slightly slower than a good, manually parallelized version.

Speculative parallelization's main advantage is that it can automatically parallelize loops of a sequential application without knowing the dependence pattern at compile time. Thus, it can obtain speedups in a parallel, multithreaded machine without the development cost of manual parallelization. This requires only some simple adjustments to the original sequential code—all within the capabilities of modern compilers. These adjustments include inserting special functions to schedule threads, performing speculative loads and stores, and commiting data when a thread successfully finishes.

Automatic parallelization is a lively research topic. Researchers are expending considerable effort to develop better compile-time analysis techniques, more sophisticated mechanisms for speculative parallelization at runtime, and more reliable theoretical models to predict whether the parallelization of a particular loop will be profitable. Its successful development would allow us to take full advantage of new hardware while running our old code.

Arturo González-Escribano is assistant professor of computer science in the Departamento de Informática, Universidad de Valladolid, Spain. Contact him at arturo@infor.uva.es.

Diego R. Llanos is associate professor of computer architecture in the Departamento de Informática, Universidad de Valladolid. Contact him at diego@infor.uva.es.

Computer welcomes your submissions to this bimonthly column. For additional information, contact Alf Weaver, the column editor, at weaver@cs.virginia.edu.