T

Thread-Level Speculation

Josep Torrellas
University of Illinois at Urbana-Champaign 4231 Siebel Center, M/C-258, Urbana, IL, USA

Synonyms

Speculative multithreading (SM); Speculative parallelization; Speculative run-time parallelization; Speculative threading; Speculative thread-level parallelization; Thread-level data speculation (TLDS); TLS

Definition

Thread-Level Speculation (TLS) refers to an environment where execution threads operate speculatively, performing potentially unsafe operations, and temporarily buffering the state they generate in a buffer or cache. At a certain point, the operations of a thread are declared to be correct or incorrect. If they are correct, the thread commits, merging the state it generated with the correct state of the program; if they are incorrect, the thread is squashed and typically restarted from its beginning. The term TLS is most often associated to a scenario where the purpose is to execute a sequential application in parallel. In this case, the compiler or the hardware breaks down the application into speculative threads that execute in parallel. However, strictly speaking, TLS can be applied to any environment where threads are executed speculatively and can be squashed and restarted.

Discussion

Basic Concepts in Thread-Level Speculation

In its most common use, Thread-Level Speculation (TLS) consists of extracting units of work (i.e., tasks) from a sequential application and executing them on different threads in parallel, hoping not to violate sequential semantics. The control flow in the sequential code imposes a relative ordering between the tasks, which is expressed in terms of predecessor and successor tasks. The sequential code also induces a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

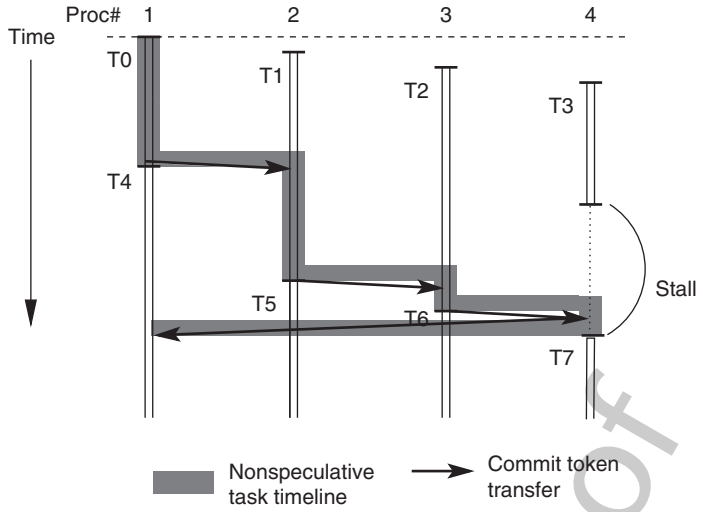
A task is *Speculative* when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. Otherwise, the task is nonspeculative. The memory accesses issued by speculative tasks are called speculative memory accesses.

When a nonspeculative task finishes execution, it is ready to *Commit*. The role of commit is to inform the rest of the system that the data generated by the task is now part of the safe, nonspeculative program state. Among other operations, committing always involves passing the *Commit Token* to the immediate successor task. This is because maintaining correct sequential semantics in the parallel execution requires that tasks commit in order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires nonspeculative status and all its predecessors have committed.

Figure 1 shows an example of several tasks running on four processors. In this example, when task T3 executing on processor 4 finishes the execution, it cannot commit until its predecessor tasks T0, T1, and T2 also finish and commit. In the meantime, depending on the hardware support, processor 4 may have to stall or may be able to start executing speculative task T7. The example also shows how the nonspeculative task status changes as tasks finish and commit, and the passing of the commit token.

Memory accesses issued by a speculative task must be handled carefully. Stores generate *Speculative Versions* of data that cannot simply be merged with the nonspeculative state of the program. The reason is that they may be incorrect. Consequently, these versions

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,



Thread-Level Speculation. Fig. 1 A set of tasks executing on four processors. The figure shows the nonspeculative task timeline and the transfer of the commit token

are stored in a *Speculative Buffer* local to the processor running the task – e.g., the first-level cache. Only when the task becomes nonspeculative are its versions safe.

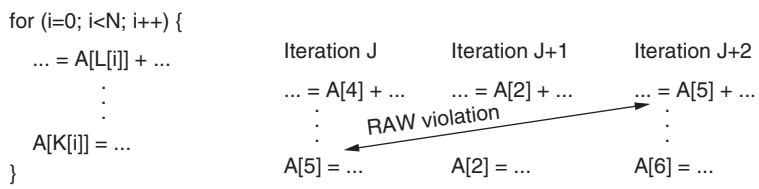
Loads issued by a speculative task try to find the requested datum in the local speculative buffer. If they miss, they fetch the correct version from the memory subsystem, i.e., the closest predecessor version from the speculative buffers of other tasks. If no such version exists, they fetch the datum from memory.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware or software support that tracks, for each individual task, the data that the task wrote and the data that the task read without first writing it. A data-dependence violation is flagged when a task modifies a datum that has been read earlier by a successor task. At this point, the consumer task is *squashed* and all the data versions that it has produced are discarded. Then, the task is re-executed.

Figure 2 shows an example of a data-dependence violation. In the example, each iteration of a loop is a task. Each iteration issues two accesses to an array, through an un-analyzable subscripted subscript. At run-time, iteration J writes A[5] after its successor iteration J+2 reads A[5]. This is a Read After Write (RAW) dependence that gets violated due to the parallel execution. Consequently, iteration J+2 is squashed and restarted. Ordinarily, all the successor tasks of iteration J+2 are also squashed at this time because they may have consumed versions generated by the squashed task. While it is possible to selectively squash only tasks that used incorrect data, it would involve extra complexity. Finally, as iteration J+2 re-executes, it will re-read A[5]. However, at this time, the value read will be the version generated by iteration J.

Note that WAR and WAW dependence violations do not need to induce task squashes. The successor task has prematurely written the datum, but the datum remains buffered in its speculative buffer. A subsequent read from a predecessor task (in a WAR violation) will get a correct version, while a subsequent write from a predecessor task (in a WAW violation) will generate a version that will be merged with main memory before the one from the successor task.

However, many proposed TLS schemes, to reduce hardware complexity, induce squashes in a variety of situations. For instance, if the system has no support to keep different versions of the same datum in different speculative buffers in the machine, cross-task WAR and WAW dependence violations induce squashes. Moreover, if the system only tracks accesses on a per-line basis, it cannot disambiguate accesses to different words in the same memory line. In this case, false sharing of a cache line by two different processors can appear as a data-dependence violation and also trigger a squash.



Thread-Level Speculation. Fig. 2 Example of a data-dependence violation

Finally, while TLS can be applied to various code structures, it is most often applied to loops. In this case, tasks are typically formed by a set of consecutive iterations.

The rest of this article is organized as follows: First, the article briefly classifies TLS schemes. Then, it describes the two major problems that any TLS scheme has to solve, namely, buffering and managing speculative state, and detecting and handling dependence violations. Next, it describes the initial efforts in TLS, other uses of TLS, and machines that use TLS.

Classification of Thread-Level Speculation Schemes

There have been many proposals of TLS schemes. They can be broadly classified depending on the emphasis on hardware versus software, and the type of target machine.

The majority of the proposed schemes use hardware support to detect cross-task dependence violations that result in task squashes (e.g., [1, 4, 6, 8, 11, 12, 14, 16, 18, 20, 23, 27, 28, 31, 32, 36]). Typically, this is attained by using the hardware cache coherence protocol, which sends coherence messages between the caches when multiple processors access the same memory line. Among all these hardware-based schemes, the majority rely on a compiler or a software layer to identify and prepare the tasks that should be executed in parallel. Consequently, there have been several proposals for TLS compilers (e.g., [9, 19, 33, 34]). Very few schemes rely on the hardware to identify the tasks (e.g., [1]).

Several schemes, especially in the early stages of TLS research, proposed software-only approaches to TLS (e.g., [7, 13, 25, 26]). In this case, the compiler typically generates code that causes each task to keep shadow locations and, after the parallel execution, checks if multiple tasks have updated a common location. If they have, the original state is restored.

Most proposed TLS schemes target small shared-memory machines of about two to eight processors (e.g., [14, 18, 27, 29]). It is in this range of parallelism that TLS is most cost effective. Some TLS proposals have focused on smaller machines and have extended a superscalar core with some hardware units that execute threads speculatively [1, 20]. Finally, some TLS proposals have targeted scalable multiprocessors [4, 23, 28]. This is a more challenging environment, given the longer communication latencies involved. It requires applications that have significant parallelism that cannot be analyzed statically by the compiler.

Buffering and Managing Speculative State

The state produced by speculative tasks is unsafe, since such tasks may be squashed. Therefore, any TLS scheme must be able to identify such state and, when necessary, separate it from the rest of the memory state. For this, TLS systems use structures, such as caches [4, 6, 12, 18, 28], and special buffers [8, 14, 23, 32], or undo logs [7, 11, 36]. This section outlines the challenges in buffering and managing speculative state. A more detailed analysis and a taxonomy is presented by Garzaran et al. [10].

Multiple Versions of the Same Variable in the System

Every time that a task writes for the first time to a variable, a new version of the variable appears in the system. Thus, two speculative tasks running on different processors may create two different versions of the same variable [4, 12]. These versions need to be buffered separately, and special actions may need to be taken so that a reader task can find the correct version out of the several coexisting in the system. Such a version will be the version created by the producer task that is the closest predecessor of the reader task.

A task has at most a single version of any given variable, even if it writes to the variable multiple times.

The reason is that, on a dependence violation, the whole task is undone. Therefore, there is no need to keep intermediate values of the variable.

Multiple Speculative Tasks per Processor

When a processor finishes executing a task, the task may still be speculative. If the TLS buffering support is such that the processor can only hold state from a single speculative task, the processor stalls until the task commits. However, to better tolerate task load imbalance, the local buffer may have been designed to buffer state from several speculative tasks, enabling the processor to execute another speculative task. In this case, the state of each task must be tagged with the ID of the task.

Multiple Versions of the Same Variable in a Single Processor

When a processor buffers state from multiple speculative tasks, it is possible that two such tasks create two versions of the same variable. This occurs in load-imbalanced applications that exhibit private data patterns (i.e., WAW dependences between tasks). In this case, the buffer will have to hold multiple versions of the same variable. Each version will be tagged with a different task ID. This support introduces complication to the buffer or cache. Indeed, on an external request, extra comparisons will need to be done if the cache has two versions of the same variable.

Merging of Task State

The state produced by speculative tasks is typically merged with main memory at task commit time; however, it can instead be merged as it is being generated. The first approach is called *Architectural Main Memory (AMM)* or *Lazy Version Management*; the second one is called *Future Main Memory (FMM)* or *Eager Version Management*. These schemes differ on whether the main memory contains only safe data (AMM) or it can also contain speculative data (FMM).

In AMM systems, all speculative versions remain in caches or buffers that are kept separate from the coherent memory state. Only when a task becomes non-speculative can its buffered state be merged with main memory. In a straightforward implementation, when a task commits, all the buffered dirty cache lines are merged with main memory, either by writing back the lines to memory [4] or by requesting ownership for them to obtain coherence with main memory [28].

In FMM systems, versions from speculative tasks are merged with the coherent memory when they are generated. However, to enable recovery from task squashes, when a task generates a speculative version of a variable, the previous version of the variable is saved in a log. Note that, in both approaches, the coherent memory state can temporarily reside in caches, which function in their traditional role of extensions of main memory.

Detecting and Handling Dependence Violations

Basic Concepts

The second aspect of TLS involves detecting and handling dependence violations. Most TLS proposals focus on data dependences, rather than control dependences. To detect (cross-task) data-dependence violations, most TLS schemes use the same approach. Specifically, when a speculative task writes a datum, the hardware sets a Speculative Write bit associated with the datum in the cache; when a speculative task reads a datum before it writes to it (an event called *Exposed Read*), the hardware sets an Exposed Read bit. Depending on the TLS scheme supported, these accesses also cause a tag associated with the datum to be set to the ID of the task.

In addition, when a task writes a datum, the cache coherence protocol transaction that sends invalidations to other caches checks these bits. If a successor task has its Exposed Read bit set for the datum, the successor task has prematurely read the datum (i.e., this is a RAW dependence violation), and is squashed [18].

If the Speculative Write and Exposed Read bits are kept on a per-word basis, only dependences on the same word can cause squashes. However, keeping and maintaining such bits on a per-word basis in caches, network messages, and perhaps directory modules is costly in hardware. Moreover, it does not come naturally to the coherence protocol of multiprocessors, which operate at the granularity of memory lines.

Keeping these bits on a per-line basis is cheaper and compatible with mainstream cache coherence protocols. However, the hardware cannot then disambiguate accesses at word level. Furthermore, it cannot combine different versions of a line that have been updated in different words. Consequently, cross-task RAW and WAW

violations, on both the same word and different words of a line (i.e., false sharing), cause squashes.

Task squash is a very costly operation. The cost is threefold: overhead of the squash operation itself, loss of whatever correct work has already been performed by the offending task and its successors, and cache misses in the offending task and its successors needed to reload state when restarting. The latter overhead appears because, as part of the squash operation, the speculative state in the cache is invalidated. Figure 3a shows an example of a RAW violation across tasks *i* and *i+j+1*. The consumer task and its successors are squashed.

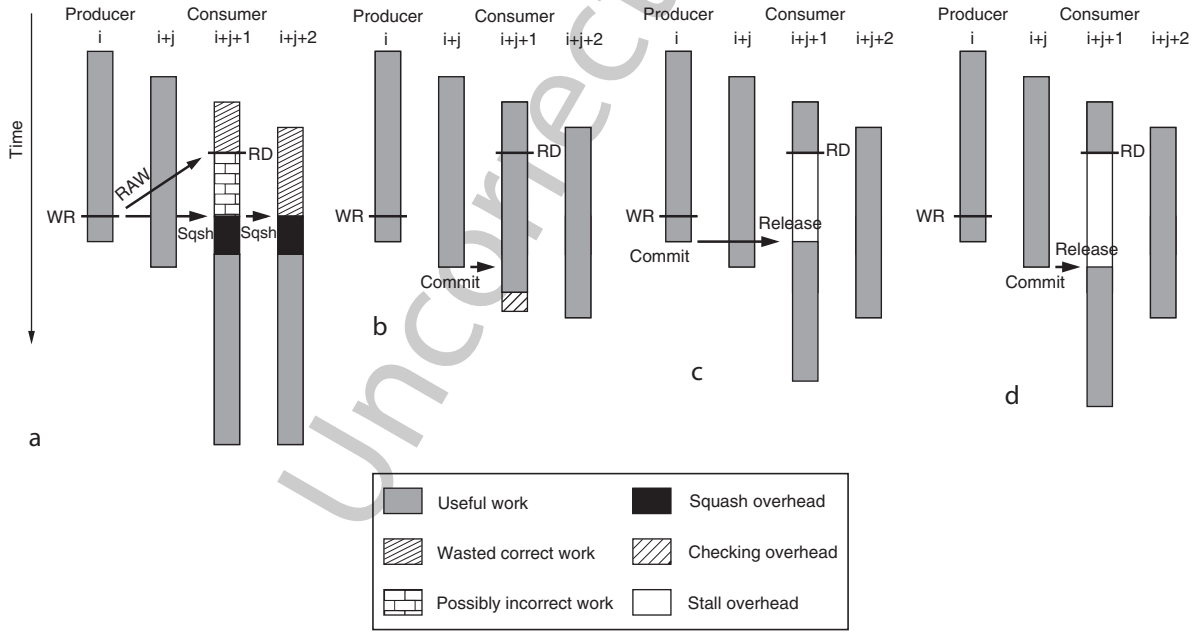
Techniques to Avoid Squashes

Since squashes are so expensive, there are techniques to avoid them. If the compiler can conclude that a certain pair of accesses will frequently cause a data-dependence violation, it can statically insert a synchronization operation that forces the correct task ordering at runtime.

Alternatively, the machine can have hardware support that records, at runtime, where dependence violations occur. Such hardware may record the program counter of the read or writes involved, or the address of the memory location being accessed. Based on this information, when these program counters are reached or the memory location is accessed, the hardware can try one of several techniques to avoid the violation. This section outlines some of the techniques that can be used. A more complete description of the choices is presented by Cintra and Torrellas [5]. Without loss of generality, a RAW violation is assumed.

Based on past history, the predictor may predict that the pair of conflicting accesses are engaged in false sharing. In this case, it can simply allow the read to proceed and then the subsequent write to execute silently, without sending invalidations. Later, before the consumer task is allowed to commit, it is necessary to check whether the sections of the line read by the consumer overlap with the sections of the line written by the producer. This can be easily done if the caches have per-word access bits. If there is no overlap, it was false sharing and the squash is avoided. Figure 3b shows the resulting time line.

When there is a true data dependence between tasks, a squash can be avoided with effective use of value prediction. Specifically, the predictor can predict the value that the producer will produce, speculatively provide it to the consumer's read, and let the consumer proceed.



Thread-Level Speculation. Fig. 3 RAW data-dependence violation that results in a squash (**a**) or that does not cause a squash due to false sharing or value prediction (**b**), or consumer stall (**c** and **d**)

Again, before the consumer is allowed to commit, it is necessary to check that the value provided was correct. The timeline is also shown in Fig. 3b.

In cases where the predictor is unable to predict the value, it can avoid the squash by stalling the consumer task at the time of the read. This case can use two possible approaches. An aggressive approach is to release the consumer task and let it read the current value as soon as the predicted producer task commits. The time line is shown in Fig. 3c. In this case, if an intervening task between the first producer and the consumer later writes the line, the consumer will be squashed. A more conservative approach is not to release the consumer task until it becomes nonspeculative. In this case, the presence of multiple predecessor writers will not squash the consumer. The time line is shown in Fig. 3d.

Initial Efforts in Thread-Level Speculation

An early proposal for hardware support for a form of speculative parallelization was made by Knight [16] in the context of functional languages. Later, the Multiscalar processor [27] was the first proposal to use a form of TLS within a single-chip multithreaded architecture. A software-only form of TLS was proposed in the LRPD test [25]. Early proposals of hardware-based TLS include the work of several authors [14, 17, 21, 29, 35].

Other Uses of Thread-Level Speculation

TLS concepts have been used in environments that have goals other than trying to parallelize sequential programs. For example, they have been used to speed up explicitly parallel programs through Speculative Synchronization [22], or for parallel program debugging [24] or program monitoring [37]. Similar concepts to TLS have been used in systems supporting hardware transactional memory [15] and continuous atomic-block operation [30].

Machines that Use Thread-Level Speculation

Several machines built by computer manufacturers have hardware support for some form of TLS – although the specific implementation details are typically not disclosed. Such machines include systems designed for Java applications such as Sun Microsystems' MAJC chip [31] and Azul Systems' Vega processor [2]. The most high-profile system with hardware support for speculative threads is Sun Microsystems' ROCK processor [3]. Other manufacturers are rumored to be developing prototypes with similar hardware.

Related Entries

►Instruction-Level Speculation
►Speculative Synchronization
►Transactional Memory

Bibliography

1. Akkary H, Driscoll M (1998) A dynamic multithreading processor. In: International symposium on microarchitecture, Dallas, November 1998
2. Azul Systems. Vega 3 Processor. http://www.azulsystems.com/products/vega/processor
3. Chaudhry S, Cypher R, Ekman M, Karlsson M, Landin A, Yip S, Zeffer H, Tremblay M (2009) Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's ROCK Processor. In: International symposium on computer architecture, Austin, June 2009
4. Cintra M, Martínez JF, Torrellas J (2000) Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: International symposium on computer architecture, Vancouver, June 2000, pp 13–24
5. Cintra M, Torrellas J (2002) Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In: Proceedings of the 8th High-Performance computer architecture conference, Boston, Feb 2002
6. Figueiredo R, Fortes J (2001) Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocesors. In: Proceedings of the international conference on parallel processing, Valencia, Spain, September 2001
7. Frank M, Lee W, Amarasinghe S (2001) A software framework for supporting general purpose applications on raw computation fabrics. Technical report, MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001
8. Franklin M, Sohi G (1996) ARB: a hardware mechanism for dynamic reordering of memory references. IEEE Trans Comput 45(5):552–571
9. Garcia C, Madriles C, Sanchez J, Marcuello P, Gonzalez A, Tullsen D (2005) Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In: Conference on programming language design and implementation, Chicago, Illinois, June 2005
10. Garzarán M, Prvulovic M, Llabería J, Viñals V, Rauchwerger L, Torrellas J (2005) Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. ACM Trans Archit Code Optim
11. Garzaran MJ, Prvulovic M, Llabería JM, Viñals V, Rauchwerger L, Torrellas J (2003) Using software logging to support multi-version buffering in thread-level speculation. In: International

AU1

conference on parallel architectures and compilation techniques, New Orleans, Sept 2003
12. Gopal S, Vijaykumar T, Smith J, Sohi G (1998) Speculative versioning cache. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
13. Gupta M, Nim R (1998) Techniques for speculative run-time parallelization of loops. In: Proceedings of supercomputing 1998, ACM Press, Melbourne, Australia, Nov 1998
14. Hammond L, Willey M, Olukotun K (1998) Data speculation support for a chip multiprocessor. In: International conference on architectural support for programming languages and operating systems, San Jose, California, Oct 1998, pp 58–69
15. Herlihy M, Moss E (1993) Transactional memory: architectural support for lock-free data structures. In: International symposium on computer architecture, IEEE Computer Society Press, San Diego, May 1993
16. Knight T (1986) An architecture for mostly functional languages. In: ACM lisp and functional programming conference, ACM Press, New York, Aug 1986, pp 500–519
17. Krishnan V, Torrellas J (1998) Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In: International conference on supercomputing, Melbourne, Australia, July 1998
18. Krishnan V, Torrellas J (1999) A chip-multiprocessor architecture with speculative multithreading. IEEE Trans Comput 48(9):866–880
19. Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J (2006) POSH: A TLS compiler that exploits program structure. In: International symposium on principles and practice of parallel programming, San Diego, Mar 2006
20. Marcuello P, Gonzalez A (1999) Clustered speculative multithreaded processors. In: International conference on supercomputing, Rhodes, Island, June 1999, pp 365–372
21. Marcuello P, Gonzalez A, Tubella J (1998) Speculative multithreaded processors. In: International conference on supercomputing, ACM, Melbourne, Australia, July 1998
22. Martinez J, Torrellas J (2002) Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
23. Prvulovic M, Garzaran MJ, Rauchwerger L, Torrellas J (2001) Removing architectural bottlenecks to the scalability of speculative parallelization. In: Proceedings of the 28th international symposium on computer architecture (ISCA'01), New York, June 2001, pp 204–215
24. Prvulovic M, Torrellas J (2003) ReEnact: using thread-level speculation to debug data races in multithreaded codes. In: International symposium on computer architecture, San Diego, June 2003
25. Rauchwerger L, Padua D (1995) The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Conference on programming language design and implementation, La Jolla, California, June 1995
26. Rundberg P, Stenstrom P (2000) Low-cost thread-level data dependence speculation on multiprocessors. In: Fourth workshop on multithreaded execution, architecture and compilation, Monterrey, Dec 2000
27. Sohi G, Breach S, Vijaykumar T (1995) Multiscalar processors. In: International Symposium on computer architecture, ACM Press, New York, June 1995
28. Steffan G, Colohan C, Zhai A, Mowry T (2000) A scalable approach to thread-level speculation. In: Proceedings of the 27th Annual International symposium on computer architecture, Vancouver, June 2000, pp 1–12
29. Steffan G, Mowry TC (1998) The potential for using thread-level data speculation to facilitate automatic parallelization. In: International symposium on high-performance computer architecture, Las Vegas, Feb 1998
30. Torrellas J, Ceze L, Tuck J, Cascaval C, Montesinos P, Ahn W, Prvulovic M (2009) The bulk multicore architecture for improved programmability. Communications of the ACM, New York
31. Tremblay M (1999) MAJC: microprocessor architecture for java computing. Hot Chips, Palo Alto, Aug 1999
32. Tsai J, Huang J, Amlo C, Lilja D, Yew P (1999) The superthreaded processor architecture. IEEE Trans Comput 48(9):881–902
33. Vijaykumar T, Sohi G (1998) Task selection for a multiscalar processor. In: International symposium on microarchitecture, Dallas, Nov 1998, pp 81–92
34. Zhai A, Colohan C, Steffan G, Mowry T (2002) Compiler optimization of scalar value communication between speculative threads. In: International conference on architectural support for programming languages and operating systems, San Jose, Oct 2002
35. Zhang Y, Rauchwerger L, Torrellas J (1998) Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In: Proceedings of the 4th International symposium on high-performance computer architecture (HPCA), Phoenix, Feb 1998, pp 162–174
36. Zhang Y, Rauchwerger L, Torrellas J (1999) Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In: Proceedings of the 5th international symposium on high-performance computer architecture, Orlando, Jan 1999, pp 135–139
37. Zhou P, Qin F, Liu W, Zhou Y, Torrellas (2004) iWatcher: efficient architectural support for software debugging. In: International symposium on computer architecture, IEEE Computer society, München, June 2004